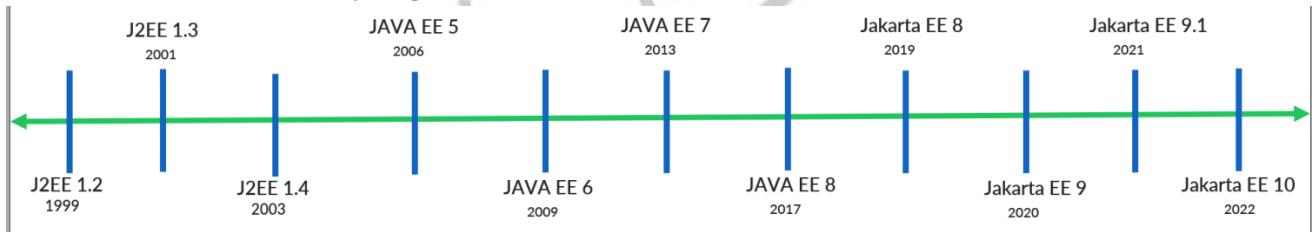


Section 1: Introduction to Spring Framework

1. What is Spring ?

- Spring Framework (viết tắt là Spring) là một framework phát triển, mạnh mẽ và rất linh hoạt, tập trung vào việc xây dựng các ứng dụng web bằng Java.
- Spring giúp lập trình Java nhanh hơn, dễ dàng hơn và an toàn hơn cho mọi người. Nó tập trung vào tốc độ, sự đơn giản và năng suất đã khiến nó trở thành khung công tác Java phổ biến nhất thế giới.
- Cho dù bạn đang xây dựng các microservice an toàn, tương tác, dựa trên đám mây cho web hay các luồng dữ liệu truyền trực tuyến phức tạp cho doanh nghiệp, thì Spring đều có các công cụ để trợ giúp.
- Ra đời như một giải pháp thay thế cho EJB vào đầu những năm 2000, Spring framework nhanh chóng vượt qua đối thủ nhờ sự đơn giản, nhiều tính năng và tích hợp thư viện của bên thứ ba.
- Nó phổ biến đến mức đối thủ cạnh tranh chính của nó đã bỏ cuộc khi Oracle dừng quá trình phát triển của Java EE 8 và cộng đồng đã tiếp nhận việc bảo trì nó thông qua Jakarta EE.
- Lý do chính của sự thành công của Spring framework là nó thường xuyên giới thiệu các tính năng/dự án dựa trên xu hướng thị trường mới nhất, nhu cầu của cộng đồng Dev. Ví dụ: SpringBoot
- Spring là mã nguồn mở. Nó có một cộng đồng lớn và tích cực cung cấp phản hồi liên tục dựa trên nhiều trường hợp sử dụng trong thế giới thực.

2. Jakarta EE Vs Spring



Java/Jakarta Enterprise Edition (EE) chứa Servlet, JSP, EJB, JMS, RMI, JPA, JSF, JAXB, JAX-WS, Web Sockets, v.v.

Các thành phần của Java/Jakarta Enterprise Edition (EE) như EJB, Servlet về bản chất là phức tạp do đó mọi người đều điều chỉnh Spring framework để phát triển ứng dụng web.

Java EE đã bỏ cuộc chạy đua với Spring framework, khi Oracle dừng quá trình phát triển của Java EE 8 và cộng đồng đã tiếp nhận việc bảo trì nó thông qua Jakarta EE.

Vì Oracle sở hữu nhãn hiệu cho tên "Java", Java EE được đổi tên thành Jakarta EE. Tất cả các gói đều được cập nhật bằng javax." đến jakarta." thay đổi namespace.

3. SPRING RELEASE TIMELINE

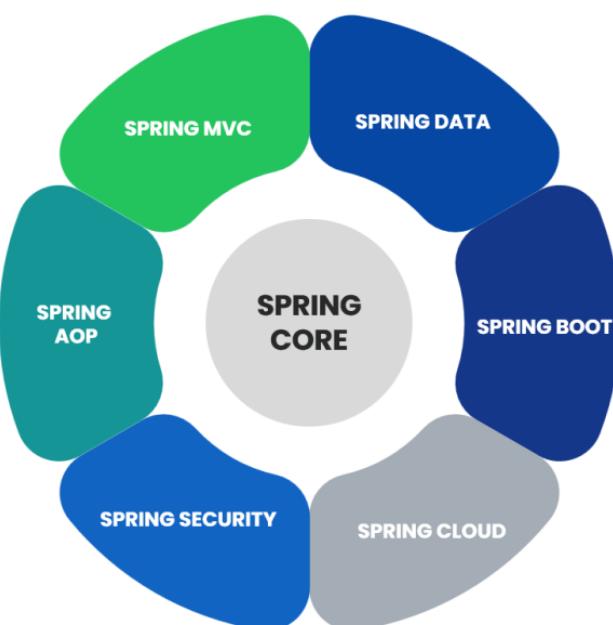


Phiên bản đầu tiên của Spring được viết bởi Rod Johnson, người đã phát hành framework cùng với việc xuất bản cuốn sách Expert One-on-One J2EE Design and Development vào tháng 10 năm 2002

Spring ra đời vào năm 2003 như một phản ứng đối với sự phức tạp của các đặc tả J2EE ban đầu. Trong khi một số người coi Java EE và Spring là đối thủ cạnh tranh, thì trên thực tế, Spring là phần bổ sung cho Java EE. Mô hình lập trình Spring không bao gồm đặc tả nền tảng Java EE; thay vào đó, nó tích hợp với các thông số kỹ thuật riêng lẻ được lựa chọn cẩn thận từ EE

Spring tiếp tục đổi mới và phát triển. Ngoài Spring Framework, còn có các dự án khác, chẳng hạn như Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, v.v.

4. Spring Core



- Spring Core là trái tim của toàn bộ Spring. Nó chứa một số lớp, nguyên tắc và cơ chế khung cơ sở.
- Toàn bộ Spring Framework và các dự án khác của Spring được phát triển trên Spring Core.
- Spring Core chứa các thành phần quan trọng sau:
 - ✓ IoC (Inversion of Control)
 - ✓ DI (Dependency Injection)
 - ✓ Beans
 - ✓ Context
 - ✓ SpEL (Spring Expression Language)
 - ✓ IoC Container

5. INVERSION OF CONTROL & DEPENDENCY INJECTION



- Inversion of Control (IoC) là Nguyên tắc thiết kế phần mềm, không phụ thuộc vào ngôn ngữ, không thực sự tạo đối tượng nhưng mô tả cách đối tượng được tạo.
- IoC là nguyên tắc, trong đó luồng điều khiển của chương trình được đảo ngược: thay vì lập trình viên kiểm soát luồng chương trình, khung hoặc dịch vụ sẽ kiểm soát luồng chương trình.
- Dependency Injection là mô hình mà Inversion of Control đạt được.
- Thông qua Dependency Injection, trách nhiệm tạo các đối tượng được chuyển từ ứng dụng sang bộ chứa Spring IoC. Nó làm giảm sự ghép nối giữa nhiều đối tượng vì nó được khung tự động đưa vào.

6. ADVANTAGES OF IoC & DI

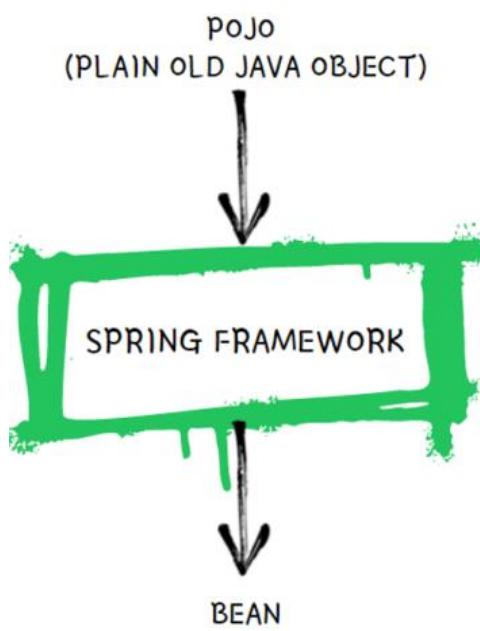
Những lợi ích của Inversion of Control (IoC) và Dependency Injection (DI):

1. Giảm sự kết nối chặt chẽ: IoC và DI giúp giảm sự kết nối chặt chẽ giữa các thành phần. Với DI, các phụ thuộc của một lớp được tiêm vào từ bên ngoài thay vì được tạo hoặc quản lý bên trong. Điều này giảm sự kết nối trực tiếp giữa các thành phần và cho phép dễ dàng bảo trì, kiểm thử và thay đổi mã nguồn.
2. Mã nguồn có tính mô-đun và có thể tái sử dụng: IoC và DI thúc đẩy tính mô-đun và khả năng tái sử dụng mã nguồn. Bằng cách tách biệt việc tạo và quản lý phụ thuộc khỏi lớp chính, các thành phần có thể dễ dàng được sử dụng lại trong các ngữ cảnh hoặc dự án khác nhau. Điều này thúc đẩy việc tổ chức mã nguồn và tạo kiến trúc mô-đun và linh hoạt hơn.
3. Khả năng kiểm thử: DI cải thiện khả năng kiểm thử. Bằng cách tiêm các phụ thuộc, bạn có thể dễ dàng cung cấp các phiên bản giả định hoặc mô phỏng của các phụ thuộc trong quá trình kiểm thử. Điều này cho phép kiểm thử đơn vị các thành phần riêng lẻ mà không cần thiết lập phức tạp hoặc phụ thuộc vào các phụ thuộc thực tế. Việc kiểm thử trở nên đơn giản và tập trung vào các đơn vị mã nguồn cụ thể.
4. Tính linh hoạt và mở rộng: IoC và DI làm cho mã nguồn của bạn linh hoạt và dễ mở rộng. Bằng cách tách biệt các thành phần, bạn có thể dễ dàng thay thế các phụ thuộc bằng các phiên bản thay thế. Điều này cho phép dễ dàng tùy chỉnh, cấu hình và mở rộng ứng dụng. Nó cho phép tích hợp các tính năng hoặc công nghệ mới mà không cần sửa đổi đáng kể mã nguồn hiện có.
5. Khuyến khích các nguyên tắc tốt nhất: IoC và DI khuyến khích việc sử dụng các nguyên tắc tốt nhất trong phát triển phần mềm, chẳng hạn như Nguyên tắc Đơn trách nhiệm (SRP) và Nguyên tắc Đảo ngược phụ thuộc (DIP). Những nguyên tắc này thúc đẩy việc phân tách quyền trách

nhiệm, trừu tượng hóa phụ thuộc và thiết kế mô-đun, dẫn đến mã nguồn sạch hơn, dễ bảo trì và có khả năng mở rộng hơn.

6. Tính dễ đọc và dễ bảo trì: Bằng cách rõ ràng khai báo các phụ thuộc và tách biệt việc quản lý chúng, DI cải thiện tính dễ đọc và dễ bảo trì của mã nguồn. Nó cung cấp thông tin rõ ràng và rõ ràng về các phụ thuộc cần thiết của một lớp, giúp các nhà phát triển dễ hiểu và chỉnh sửa mã nguồn. Điều này có thể dẫn đến sự hợp tác tốt hơn giữa các thành viên trong nhóm và việc bảo trì mã nguồn mượt mà hơn.

7. SPRING BEANS, CONTEXT, SpEL



- Bất kỳ lớp Java bình thường nào được khởi tạo, lắp ráp và quản lý bởi bộ chứa Spring IoC đều được gọi là Spring Bean.
- Các bean được tạo bằng siêu dữ liệu cấu hình mà bạn cung cấp cho container ở dạng Annotation và cấu hình XML.
- Spring IoC Container quản lý vòng đời của phạm vi Spring Bean và đưa bất kỳ phụ thuộc cần thiết nào vào bean.
- Context giống như một vị trí bộ nhớ của ứng dụng của bạn, trong đó chúng tôi thêm tất cả các phiên bản đối tượng mà chúng tôi muốn framework quản lý. Theo mặc định, Spring không biết bất kỳ đối tượng nào bạn xác định trong ứng dụng của mình. Để cho phép Spring xem các đối tượng của bạn, bạn cần thêm chúng vào context.
- SpEL cung cấp một ngôn ngữ biểu thức mạnh mẽ để truy vấn và thao tác một biểu đồ đối tượng trong thời gian chạy như cài đặt và nhận các giá trị thuộc tính, gán thuộc tính, gọi phương thức BEAN, v.v.

8. SPRING IoC CONTAINER

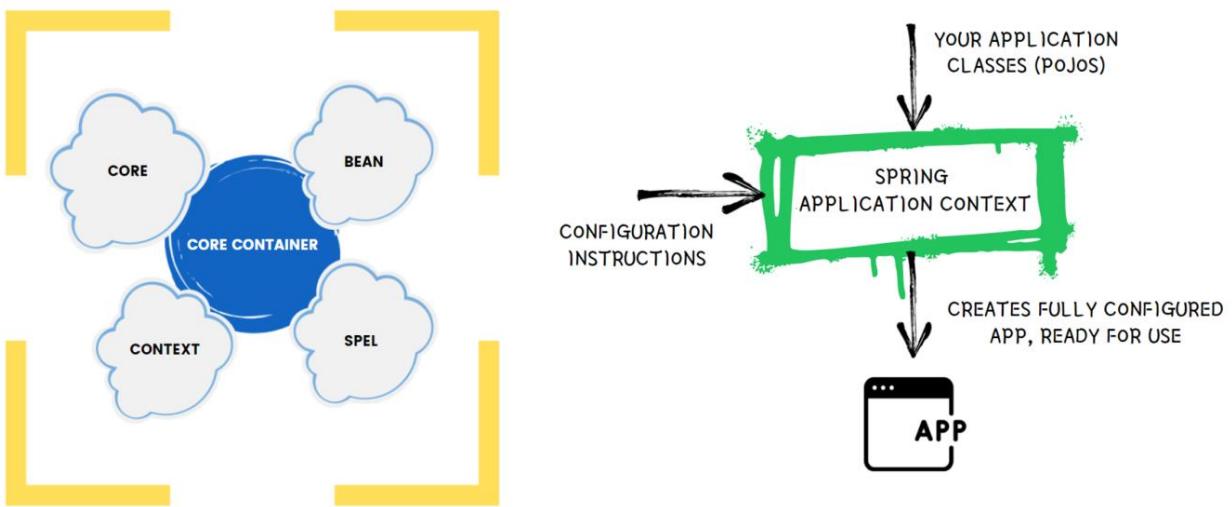
Spring IoC Container

- IoC container chịu trách nhiệm
- Để khởi tạo lớp ứng dụng
- Để cấu hình đối tượng
- Để tập hợp các phụ thuộc giữa các đối tượng

Có hai loại bộ chứa IoC. Họ đang:

- org.springframework.beans.factory.BeanFactory
- org.springframework.context.ApplicationContext

The Spring container sử dụng dependency injection (DI) để quản lý các thành phần/đối tượng tạo nên một ứng dụng.



Mars

Section 2: Creating Beans inside Spring Context

1. Maven

Apache Maven là một công cụ quản lý dự án và xây dựng phần mềm tự động trong môi trường phát triển Java. Nó giúp quản lý các phụ thuộc của dự án, quản lý vòng đời phát triển và xây dựng ứng dụng.

Maven sử dụng mô hình quản lý dự án dựa trên khái niệm Project Object Model (POM), trong đó POM là một tệp cấu hình XML mô tả cấu trúc và các yêu cầu của dự án. POM mô tả các phụ thuộc của dự án, plugin sử dụng trong quá trình xây dựng, các thiết lập và các nhiệm vụ cần thiết để xây dựng, kiểm thử và triển khai ứng dụng.

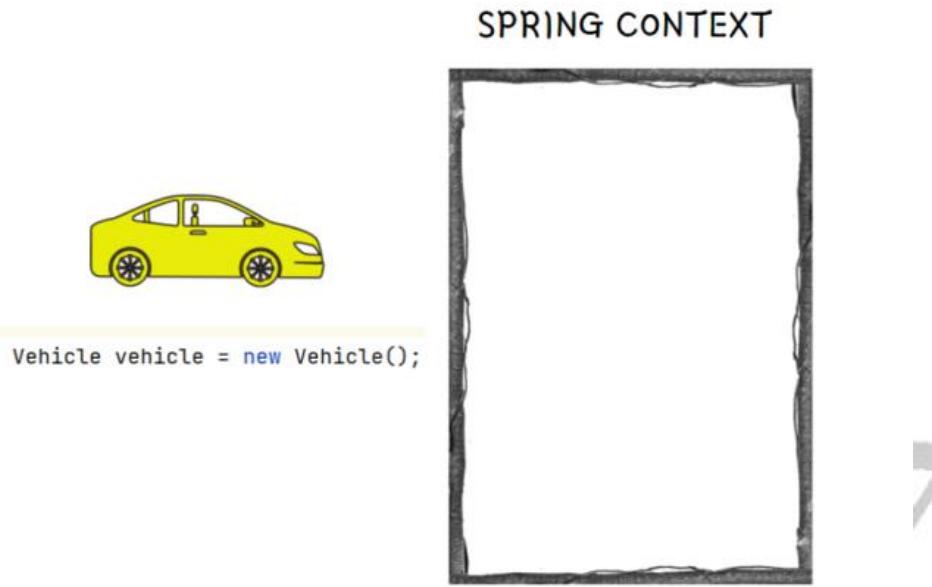
Một số tính năng chính của Maven bao gồm:

- Quản lý phụ thuộc: Maven tự động quản lý các phụ thuộc của dự án. Nó có thể tải về, cài đặt và định cấu hình các thư viện và công cụ cần thiết cho dự án, từ kho lưu trữ công cộng hoặc các kho lưu trữ nội bộ.
- Quản lý vòng đời: Maven cung cấp một chuỗi các nhiệm vụ xây dựng tiêu chuẩn, từ biên dịch mã nguồn, đóng gói tài nguyên, kiểm thử, tạo tài liệu đến triển khai ứng dụng. Nó tự động quản lý quá trình xây dựng và tạo ra sản phẩm cuối cùng.
- Plugin mở rộng: Maven hỗ trợ một loạt các plugin mở rộng cho các nhiệm vụ xây dựng khác nhau. Các plugin có thể được sử dụng để thực hiện các công việc tùy chỉnh, như đóng gói ứng dụng, chạy kiểm thử, tạo tài liệu, quản lý phiên bản, v.v.
- Tích hợp dễ dàng: Maven tích hợp tốt với các công cụ phát triển phổ biến khác như Eclipse, IntelliJ IDEA và NetBeans. Nó cung cấp các plugin và công cụ hỗ trợ để tích hợp quy trình phát triển Maven vào môi trường phát triển được sử dụng.

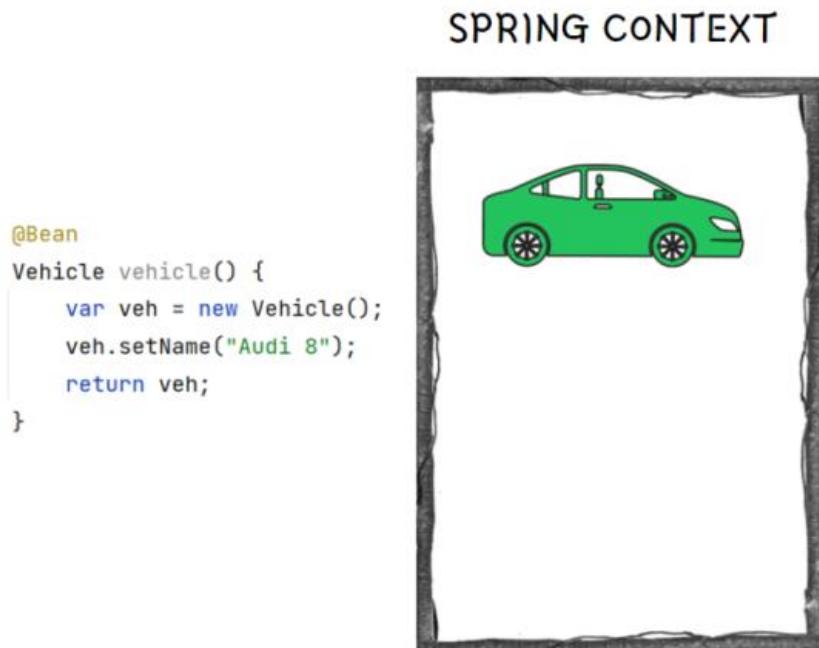
Maven là một công cụ mạnh mẽ giúp tăng hiệu suất và sự nhất quán trong quá trình phát triển phần mềm Java. Nó đơn giản hóa việc quản lý phụ thuộc, xây dựng và triển khai ứng dụng, đồng thời giúp tạo ra các dự án có cấu trúc chuẩn mà các nhà phát triển có thể dễ dàng làm việc và chia sẻ.

2. Adding new beans to spring context

- Khi tạo trực tiếp một đối tượng java với toán tử new() như hình bên dưới, thì Spring Context/Spring IoC Container của bạn sẽ không có bất kỳ manh mối nào về đối tượng.



- Annotation @Bean cho Spring biết rằng nó cần gọi phương thức này khi nó khởi tạo context và thêm object/value được trả về vào Spring context/Spring IoC container.



Ví dụ:

```
@Configuration
public class ProjectConfig {

    @Bean
    Vehicle vehicle() {
        var veh = new Vehicle();
        veh.setName("Audi 8");
        return veh;
    }
}

public class Example1 {

    public static void main(String[] args) {

        Vehicle vehicle = new Vehicle();
        vehicle.setName("Honda City");
        System.out.println("Vehicle name from non-spring context is: " +
vehicle.getName());

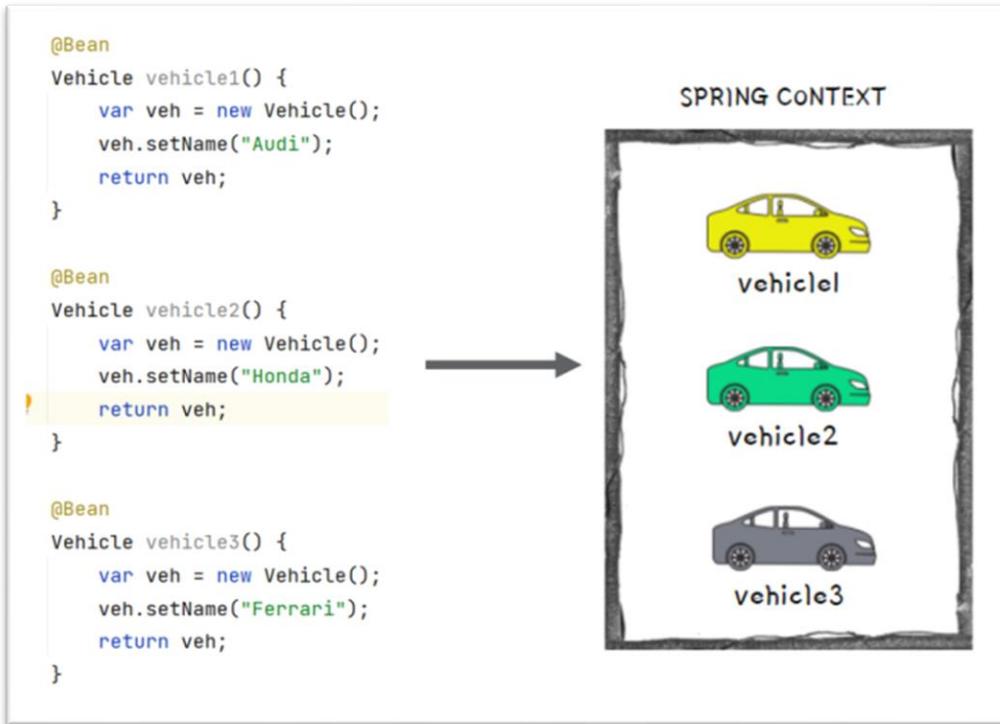
        var context = new
AnnotationConfigApplicationContext(ProjectConfig.class);

        Vehicle veh = context.getBean(Vehicle.class);
        System.out.println("Vehicle name from Spring Context is: " +
veh.getName());
    }
}

=> Kết quả trả về:
Vehicle name from non-spring context is: null
Vehicle name from Spring Context is: Audi 8
String value from Spring Context is: Hello World
Integer value from Spring Context is: 16
```

3. Understanding NoUniqueBeanDefinitionException in Spring

Khi chúng tôi tạo nhiều đối tượng cùng loại và cố gắng tìm nạp bean từ ngữ cảnh theo loại, thì Spring không thể đoán được trường hợp bạn đã khai báo mà bạn đề cập đến. Điều này sẽ dẫn đến NoUniqueBeanDefinitionException như hình bên dưới:



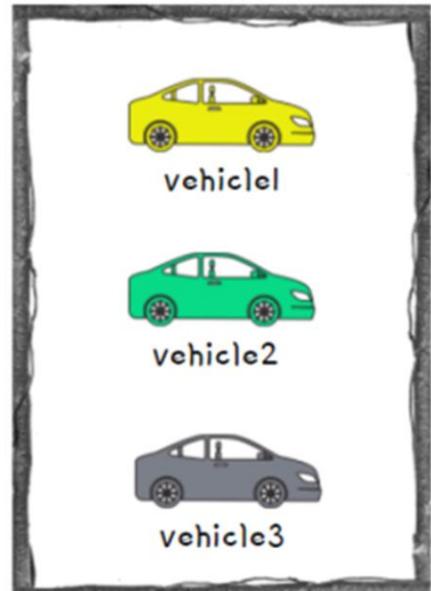
Để tránh NoUniqueBeanDefinitionException trong các tình huống này, có thể tìm nạp bean từ ngữ cảnh bằng cách nhắc đến tên của nó như được hiển thị bên dưới:

```
@Bean
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```

SPRING CONTEXT



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle veh = context.getBean("vehicle1", Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh.getName());
```



Output on Console

Vehicle name from Spring Context is: Audi

4. Providing a custom name to the bean

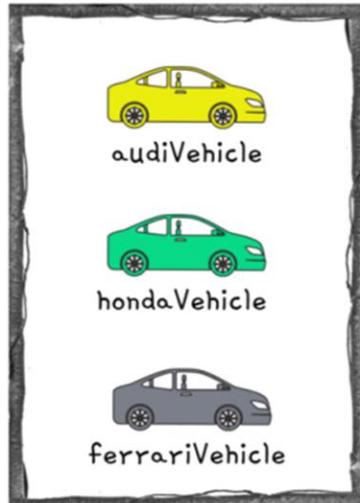
Theo mặc định, Spring sẽ coi tên phương thức là tên bean. Nhưng nếu chúng tôi có yêu cầu tùy chỉnh để xác định tên bean riêng biệt, thì chúng tôi có thể sử dụng bất kỳ phương pháp nào dưới đây với sự trợ giúp của annotation @Bean.

```
@Bean(name="audiVehicle")
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean(value="hondaVehicle")
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean("ferrariVehicle")
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```

SPRING CONTEXT



```
Vehicle veh1 = context.getBean("audiVehicle", Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh1.getName());

Vehicle veh2 = context.getBean("hondaVehicle", Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh2.getName());

Vehicle veh3 = context.getBean("ferrariVehicle", Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh3.getName());
```

Output on Console

```
Vehicle name from Spring Context is: Audi
Vehicle name from Spring Context is: Honda
Vehicle name from Spring Context is: Ferrari
```

5. Understanding @Primary Annotation inside Spring

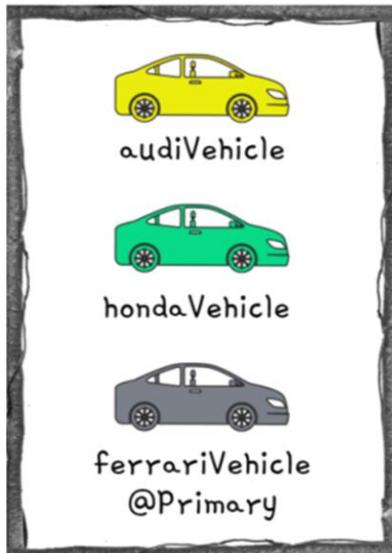
Khi bạn có nhiều bean cùng loại trong Spring context, bạn có thể đặt một trong số chúng thành chính bằng cách sử dụng annotation `@Primary`. Bean chính là cái mà Spring sẽ chọn nếu nó có nhiều tùy chọn và bạn không chỉ định tên. Nói cách khác, đó là bean mặc định mà Spring Context sẽ xem xét trong trường hợp nhầm lẫn do có nhiều bean cùng loại.

```
@Bean(name="audiVehicle")
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean(value="hondaVehicle")
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Primary
@Bean("ferrariVehicle")
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```

SPRING CONTEXT



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Primary Vehicle name from Spring Context is: " + vehicle.getName());
```

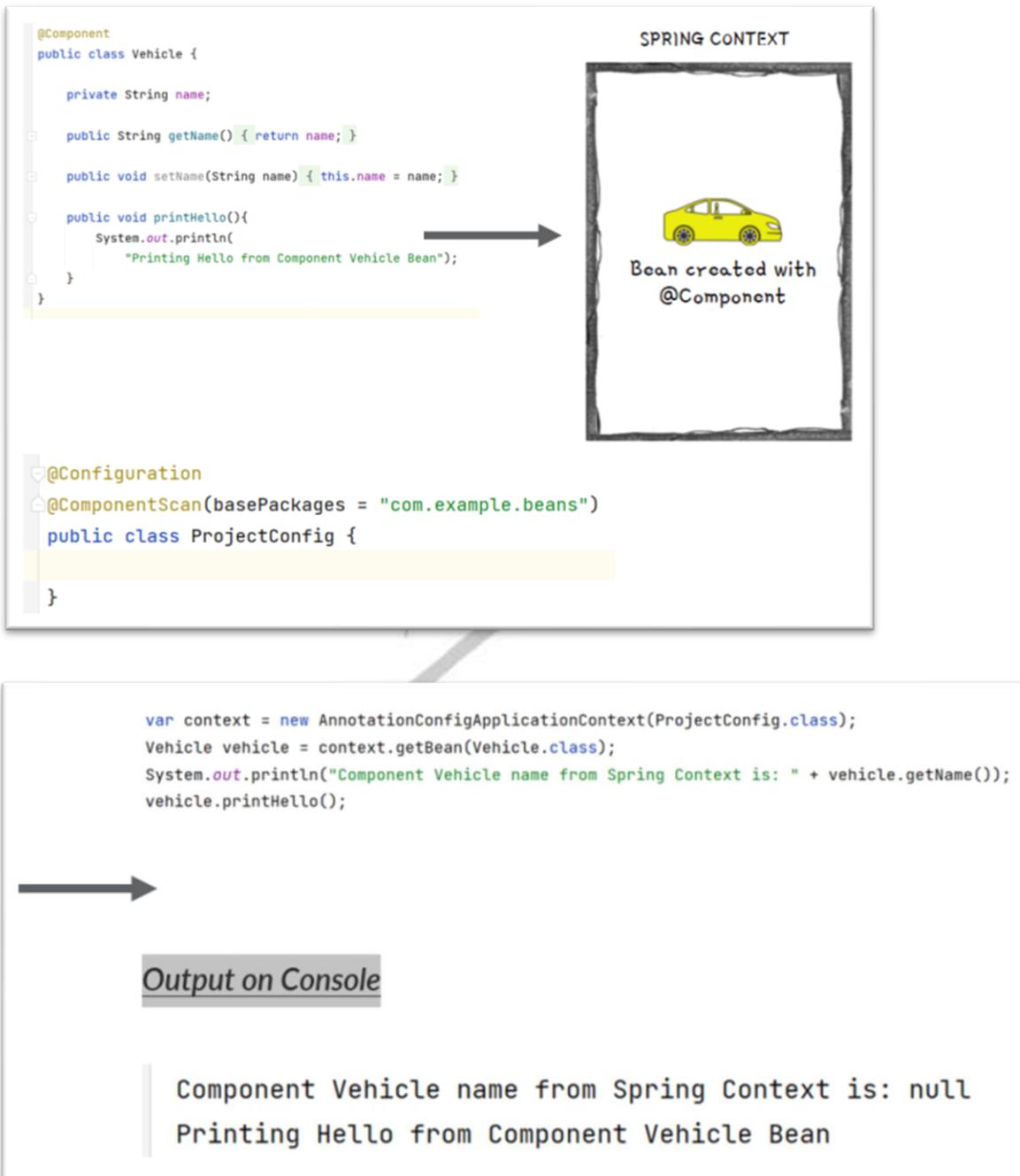
Output on Console

Primary Vehicle name from Spring Context is: Ferrari

6. Creating Beans using @Component annotation

@Component là một trong những annotation khuôn mẫu được sử dụng phổ biến nhất bởi các nhà phát triển. Sử dụng điều này, có thể dễ dàng tạo và thêm một bean vào Spring context bằng cách viết ít mã hơn so với tùy chọn @Bean. Với các annotation rập khuôn, cần thêm annotation phía trên lớp mà cần có một thể hiện trong Spring context.

Sử dụng annotation @ComponentScan trên lớp cấu hình, hướng dẫn Spring về nơi tìm các lớp mà bạn đã đánh dấu bằng annotation khuôn mẫu.



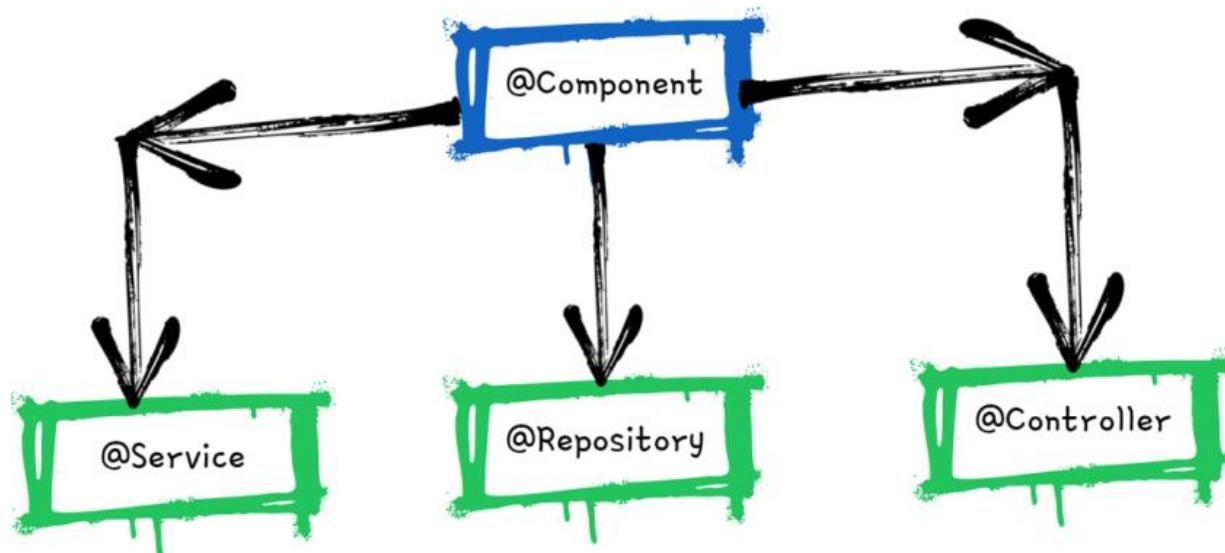
Khi một class được đánh dấu bằng `@Component`, Spring IoC Container sẽ quản lý và tạo ra một phiên bản của class đó như một bean trong quá trình khởi tạo ứng dụng. Bean này có thể được tiêm vào các thành phần khác thông qua Dependency Injection (DI).

Khi Spring IoC Container khởi tạo ứng dụng, nó sẽ quét các class và tìm các annotation `@Component` và các annotation liên quan khác để xác định các bean cần tạo và quản lý. Các bean này sẽ được đặt trong container và có thể được sử dụng trong toàn bộ ứng dụng.

7. Spring Stereotype Annotations- Chú thích khuôn mẫu spring

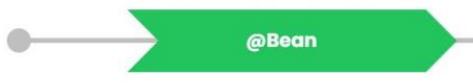
+ Spring cung cấp **các annotation** đặc biệt gọi là Stereotype annotations sẽ giúp tạo các Spring bean một cách tự động trong ngữ cảnh ứng dụng.

+ Các **Stereotype Annotations** trong spring là `@Component`, `@Service`, `@Repository` và `@Controller`



- **@Component** được sử dụng chung trên bất kỳ lớp Java nào. Nó là cơ sở cho các annotation khác.
- **@Service** có thể được sử dụng trên các lớp bên trong lớp **service**, đặc biệt là khi chúng tôi viết logic nghiệp vụ và thực hiện lệnh gọi API bên ngoài.
- **@Repository** có thể được sử dụng trên các lớp xử lý mã liên quan đến các hoạt động liên quan đến truy cập Database như Insert, Update, Delete, v.v.
- **@Controller** có thể được sử dụng trên các lớp bên trong lớp Controller của các ứng dụng MVC.

8. Comparison between @Bean Vs @Component

 @Bean	 @Component
<ul style="list-style-type: none">• Một hoặc nhiều thể hiện của lớp có thể được thêm vào Spring context• Chúng ta có thể tạo một thể hiện đối tượng của bất kỳ loại lớp nào, kể cả các thư viện hiện tại bên trong như String, v.v.• Thông thường cần viết thêm mã như các phương thức riêng biệt để tạo các phiên bản bean• Developer sẽ có toàn quyền trong việc tạo và cấu hình bean• Khung công tác mùa xuân tạo bean dựa trên các hướng dẫn và giá trị do Developer cung cấp	<ul style="list-style-type: none">• Chỉ một thể hiện của lớp có thể được thêm vào Spring context• Chúng ta chỉ có thể tạo một thể hiện đối tượng cho lớp ứng dụng được tạo bởi nhóm Dev• Các phiên bản Bean có thể được tạo với rất ít mã như sử dụng @Component ở đầu lớp• Developer sẽ không có bất kỳ quyền kiểm soát trong việc tạo và định cấu hình bean• Khung công tác mùa xuân chịu trách nhiệm tạo bean và đăng bài mà Nhà phát triển sẽ có quyền truy cập vào nó

9. Understanding @PostConstruct Annotation

Chúng tôi đã thấy rằng khi chúng tôi đang sử dụng các stereotype annotation, chúng tôi không có quyền kiểm soát trong khi tạo một bean. Nhưng điều gì sẽ xảy ra nếu muốn thực hiện một số hướng dẫn sau khi Spring tạo bean. Tương tự, có thể sử dụng annotation @PostConstruct.

@PostConstruct là một annotation được sử dụng để đánh dấu một phương thức trong một bean sẽ được thực hiện ngay sau khi việc khởi tạo bean và injection các giá trị phụ thuộc đã hoàn thành. Annotation này cho phép thực hiện các tác vụ khởi tạo sau khi bean đã được khởi tạo nhưng trước khi bean được sử dụng.

Các phương thức được đánh dấu bằng @PostConstruct phải được public và không được chứa bất kỳ tham số nào. Spring IoC Container sẽ tìm và thực hiện các phương thức này tự động.

Spring mượn annotation @PostConstruct từ Java EE.

```

@Component
public class Vehicle {

    private String name;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    @PostConstruct
    public void initialize() {
        this.name = "Honda";
    }

    public void printHello(){...}
}

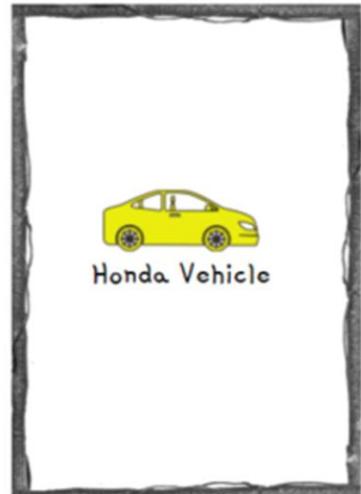
```

```

@Configuration
@ComponentScan(basePackages = "com.example.beans")
public class ProjectConfig {
}

```

SPRING CONTEXT



```

var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Component Vehicle name from Spring Context is: " + vehicle.getName());
vehicle.printHello();

```



Output on Console

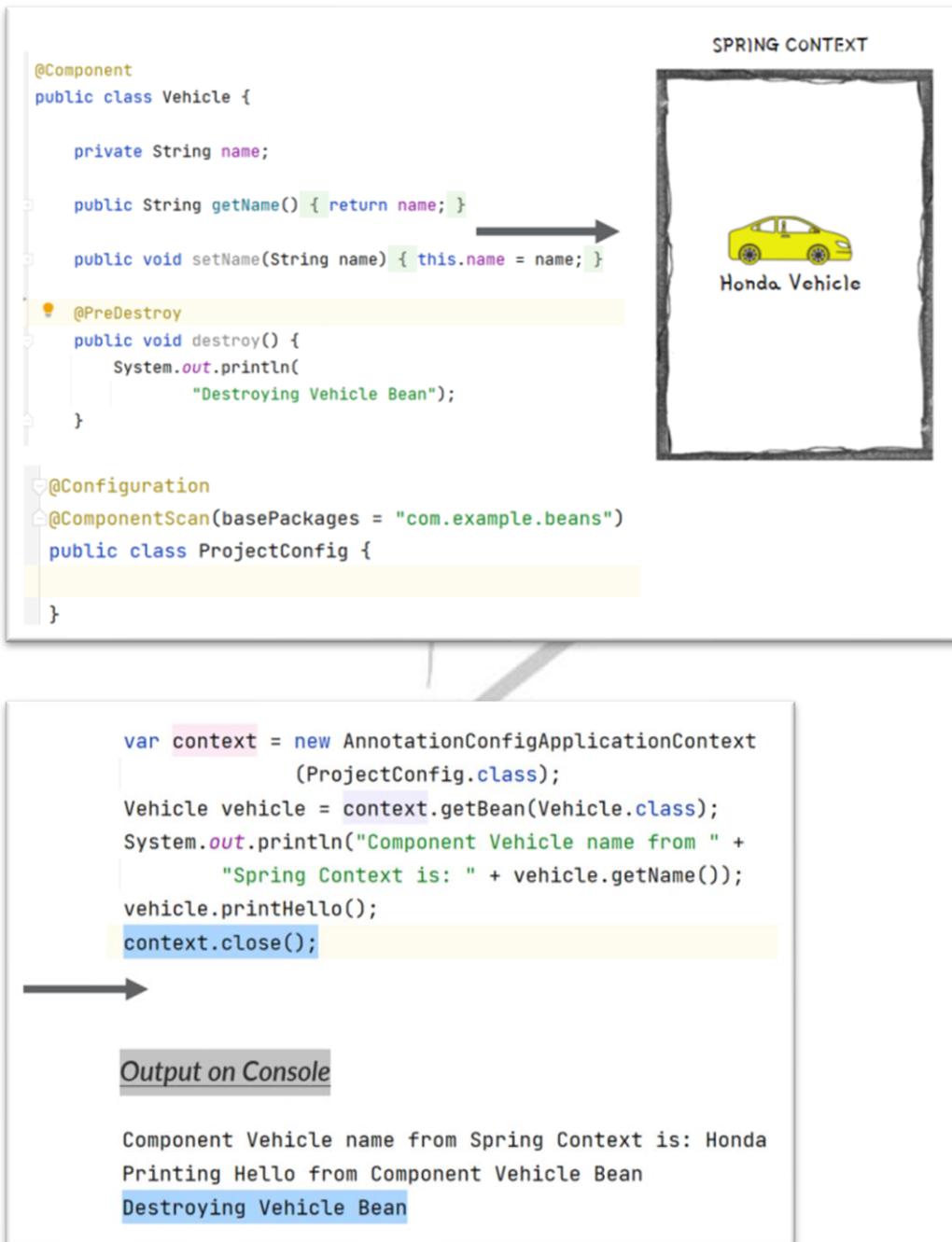
Component Vehicle name from Spring Context is: Honda
 Printing Hello from Component Vehicle Bean

10. Understanding @PreDestroy Annotation

Annotation @PreDestory có thể được sử dụng trên các phương thức và Spring sẽ đảm bảo gọi phương thức này ngay trước khi xóa và hủy context.

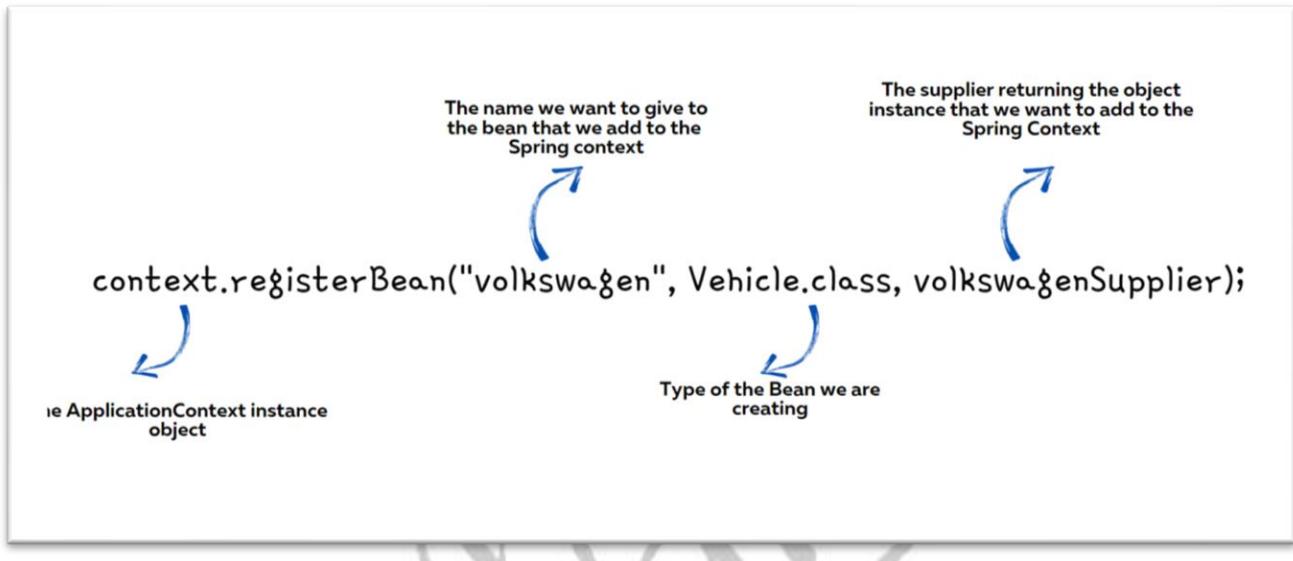
Điều này có thể được sử dụng trong các tình huống mà chúng tôi muốn đóng tài nguyên bất kỳ, kết nối Database, v.v.

Spring mượn @PreDestory annotation cũng từ Java EE.



11. Creating Beans programmatically using registerBean()

Đôi khi muốn tạo các thể hiện mới của một đối tượng và thêm chúng vào Spring context dựa trên điều kiện lập trình. Tương tự như vậy, từ phiên bản Spring 5, một cách tiếp cận mới được cung cấp để tạo các bean theo chương trình bằng cách gọi phương thức registerBean() có bên trong đối tượng ngữ cảnh.



```

public class Example7 {

    public static void main(String[] args) {

        var context = new
AnnotationConfigApplicationContext(ProjectConfig.class);

        Vehicle volkswagen = new Vehicle();
        volkswagen.setName("Volkswagen");
        Supplier<Vehicle> volkswagenSupplier = () -> volkswagen;

        Supplier<Vehicle> audiSupplier = () -> {
            Vehicle audi = new Vehicle();
            audi.setName("Audi");
            return audi;
        };

        Random random = new Random();
        int randomNumber = random.nextInt(10);
        System.out.println("randomNumber = " + randomNumber);

        if((randomNumber% 2) == 0){
            context.registerBean("volkswagen",
                Vehicle.class,volkswagenSupplier);
        }else{
            context.registerBean("audi",
                Vehicle.class,audiSupplier);
        }
        Vehicle volksVehicle = null;
        Vehicle audiVehicle = null;
        try {
            volksVehicle = context.getBean("volkswagen",Vehicle.class);
        }catch (NoSuchBeanDefinitionException noSuchBeanDefinitionException){
            System.out.println("Error while creating Volkswagen vehicle");
        }
        try {
            audiVehicle = context.getBean("audi",Vehicle.class);
        }catch (NoSuchBeanDefinitionException noSuchBeanDefinitionException){
            System.out.println("Error while creating Audi vehicle");
        }

        if(null != volksVehicle){
            System.out.println("Programming Vehicle name from Spring Context
is: " + volksVehicle.getName());
        }else{
            System.out.println("Programming Vehicle name from Spring Context
is: " + audiVehicle.getName());
        }

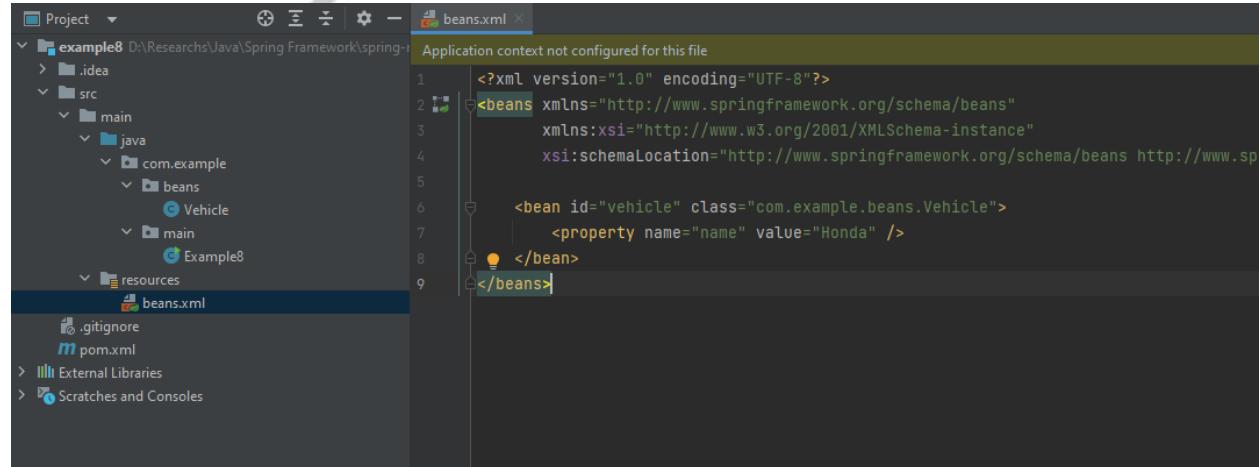
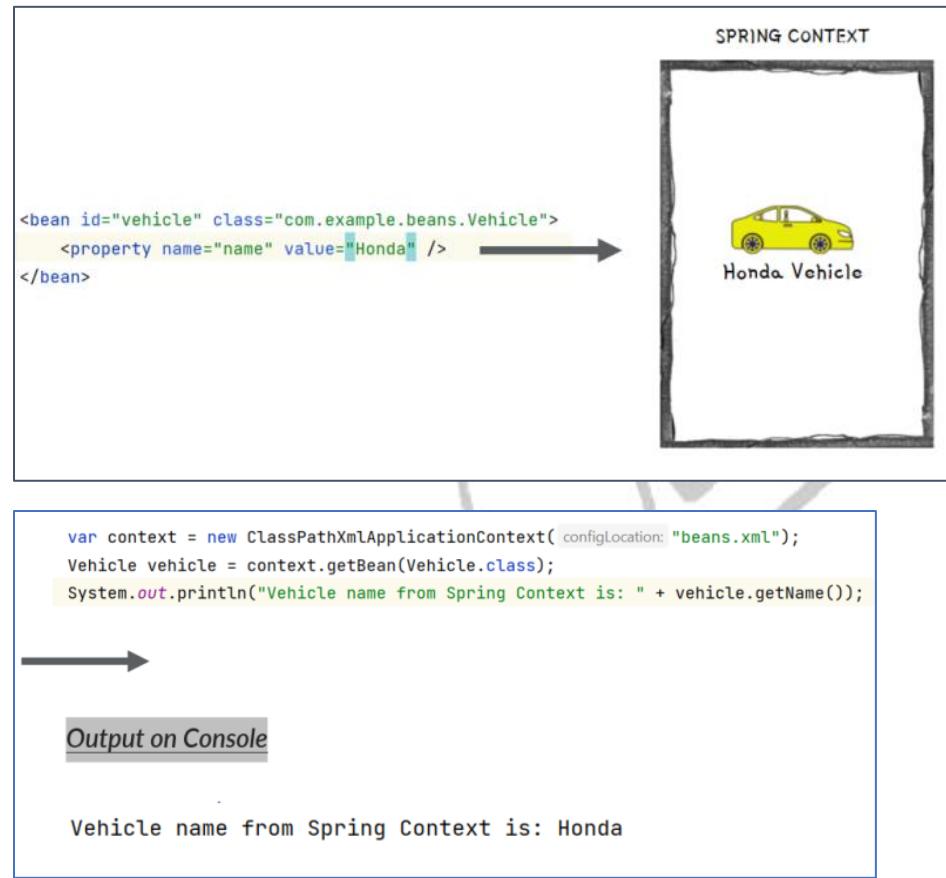
    }
}
// output
randomNumber = 5
Error while creating Volkswagen vehicle
Programming Vehicle name from Spring Context is: Audi

```

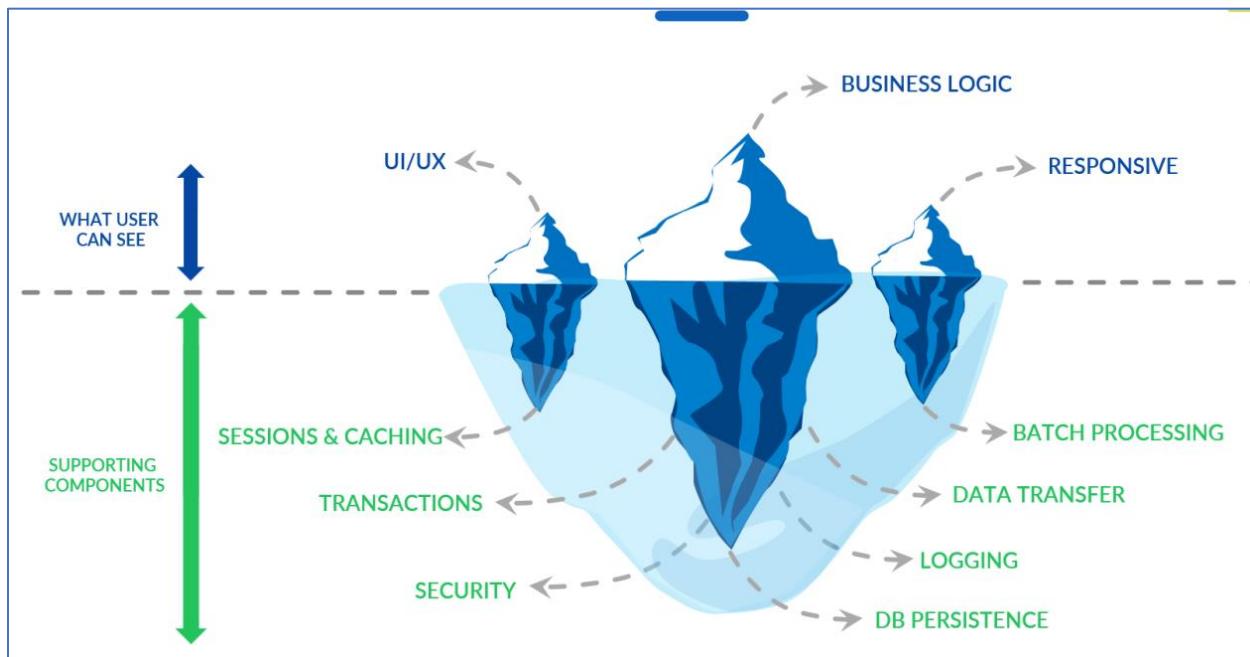
12. Creating Beans using XML Configurations (example)

Trong các phiên bản đầu tiên của Spring, bean và các cấu hình khác thường được thực hiện bằng XML. Nhưng theo thời gian, nhóm Spring mang đến các cấu hình dựa trên annotation để giúp các nhà phát triển trở nên dễ dàng. Ngày nay, chỉ có thể thấy các cấu hình XML trong các ứng dụng cũ hơn được xây dựng dựa trên các phiên bản đầu tiên của Spring.

Bạn nên hiểu cách tạo một bean bên trong Spring context bằng cách sử dụng các cấu hình kiểu XML. Vì vậy, sẽ rất hữu ích nếu có một kịch bản mà bạn cần làm việc trong một dự án dựa trên các phiên bản ban đầu của Spring.

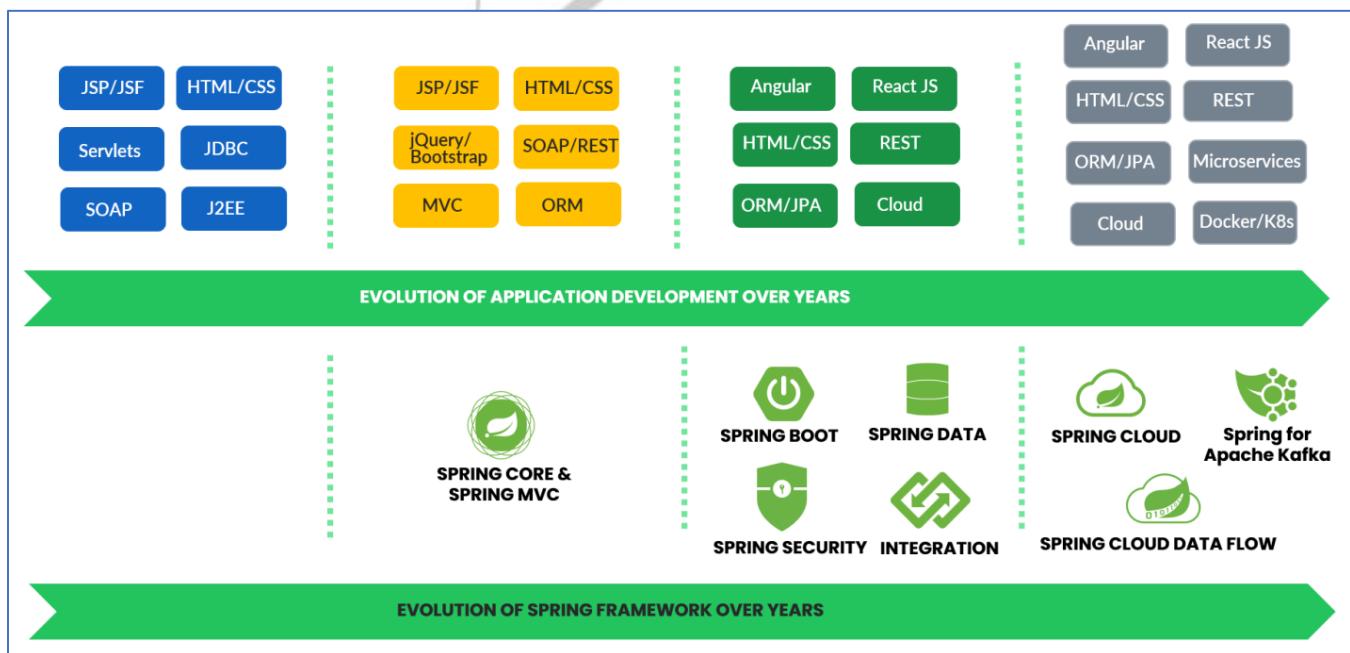
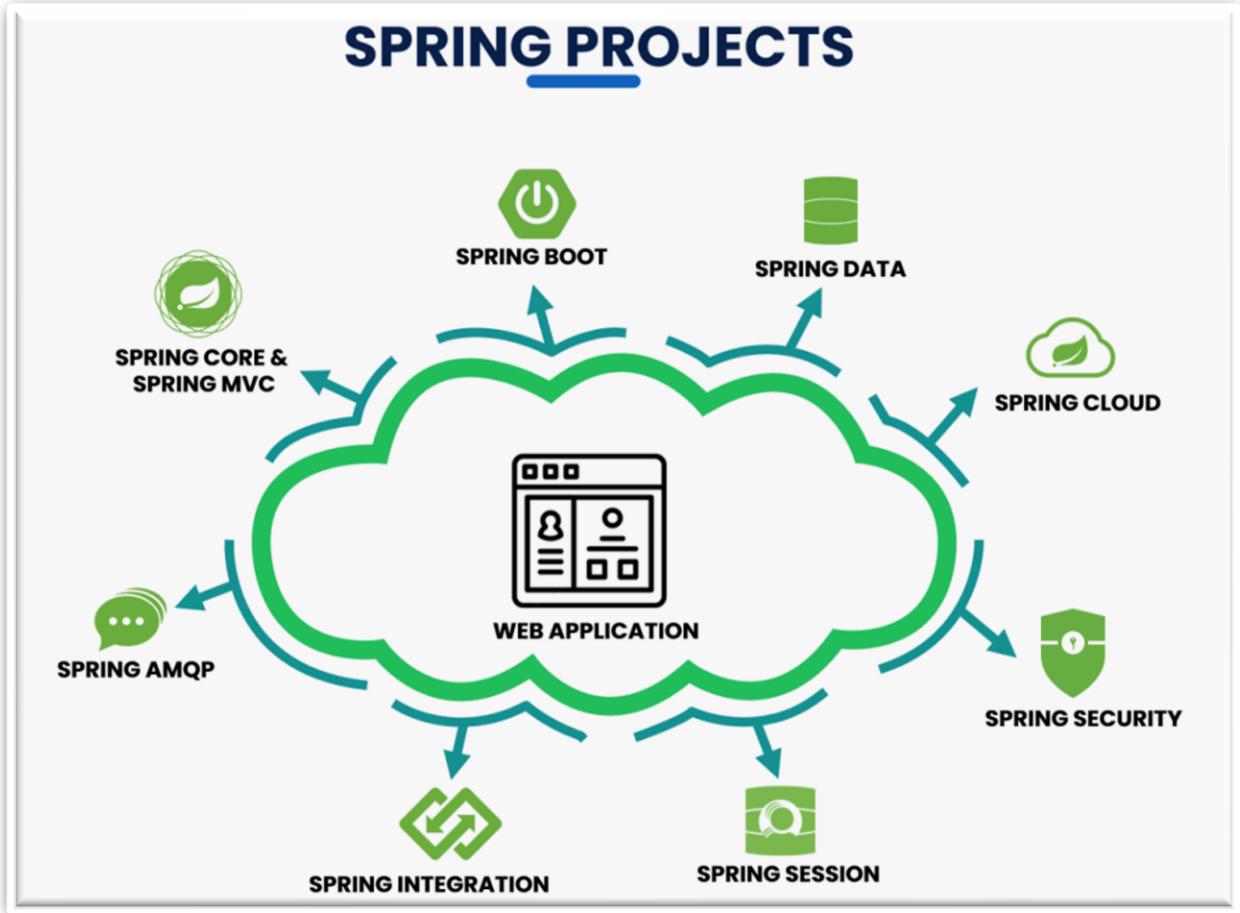


13. Why should we use frameworks



 DEV SANJEEV	 DEV VICKY
<p>Uses best readily available best frameworks like Spring, Angular etc. to build a web app</p> <ul style="list-style-type: none">• Tận dụng Security, Logging etc. from frameworks• Có thể dễ dàng mở rộng quy mô ứng dụng của mình• Ứng dụng sẽ hoạt động theo cách có thể dự đoán được• Tập trung nhiều hơn vào logic kinh doanh• Ít nỗ lực hơn và nhiều kết quả/doanh thu hơn	<p>Build his own code by himself to build a web app</p> <ul style="list-style-type: none">• Cần xây dựng mã cho Security, Logging, v.v.• Mở rộng quy mô của anh ấy không phải là một lựa chọn cho đến khi anh ấy kiểm tra mọi thứ• Ứng dụng có thể không hoạt động theo cách có thể dự đoán được• Tập trung nhiều hơn vào các thành phần hỗ trợ• Nhiều nỗ lực hơn và ít kết quả/doanh thu hơn

14. Introduction to Spring Projects



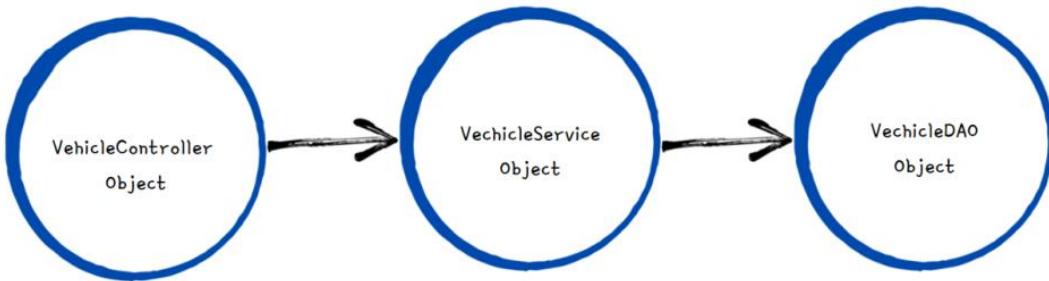
Section 3: Wiring Beans using @Autowiring

1. Introduction to wiring & auto-wiring inside Spring

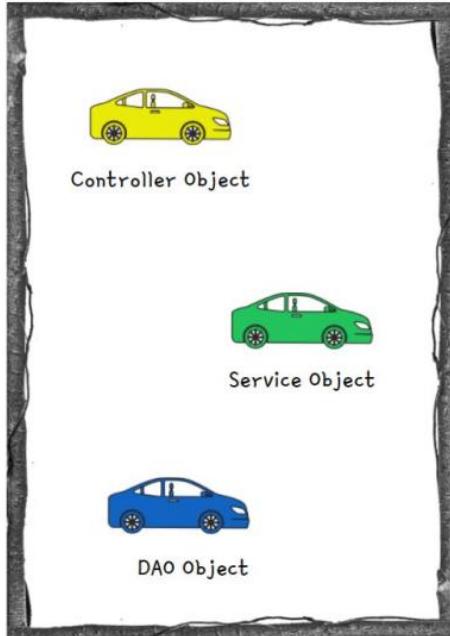
Trong Spring Framework, wiring là quá trình kết nối các thành phần (beans) với nhau để tạo ra sự phụ thuộc giữa chúng. Wiring có thể được thực hiện bằng cách chỉ định các liên kết giữa các beans trong cấu hình hoặc sử dụng tự động kết nối (auto-wiring) dựa trên quy tắc được xác định.

Wiring trong Spring có mục đích chính là đảm bảo rằng các beans có thể tương tác và làm việc với nhau một cách hợp lý. Khi các beans được kết nối, các phụ thuộc (dependencies) giữa chúng được giải quyết, cho phép chúng truy cập và sử dụng các tính năng, dịch vụ hoặc tài nguyên của nhau.

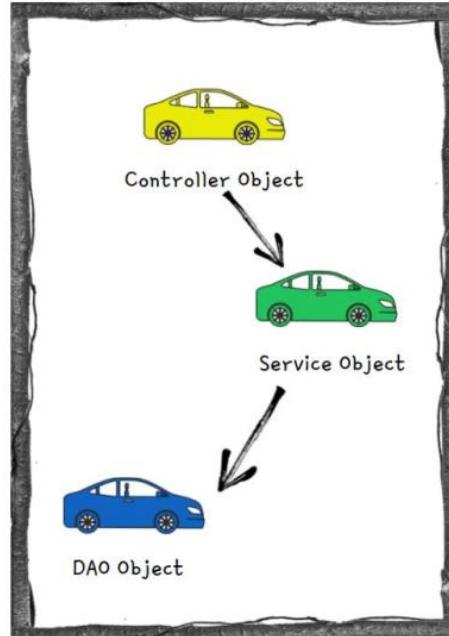
Auto-wiring là một tính năng của Spring Framework cho phép tự động kết nối các beans với nhau mà không cần chỉ định rõ ràng trong cấu hình. Spring sẽ tự động tìm kiếm các beans phù hợp dựa trên kiểu dữ liệu và tên biến và thực hiện kết nối tự động.



SPRING CONTEXT WITH OUT
WIRING

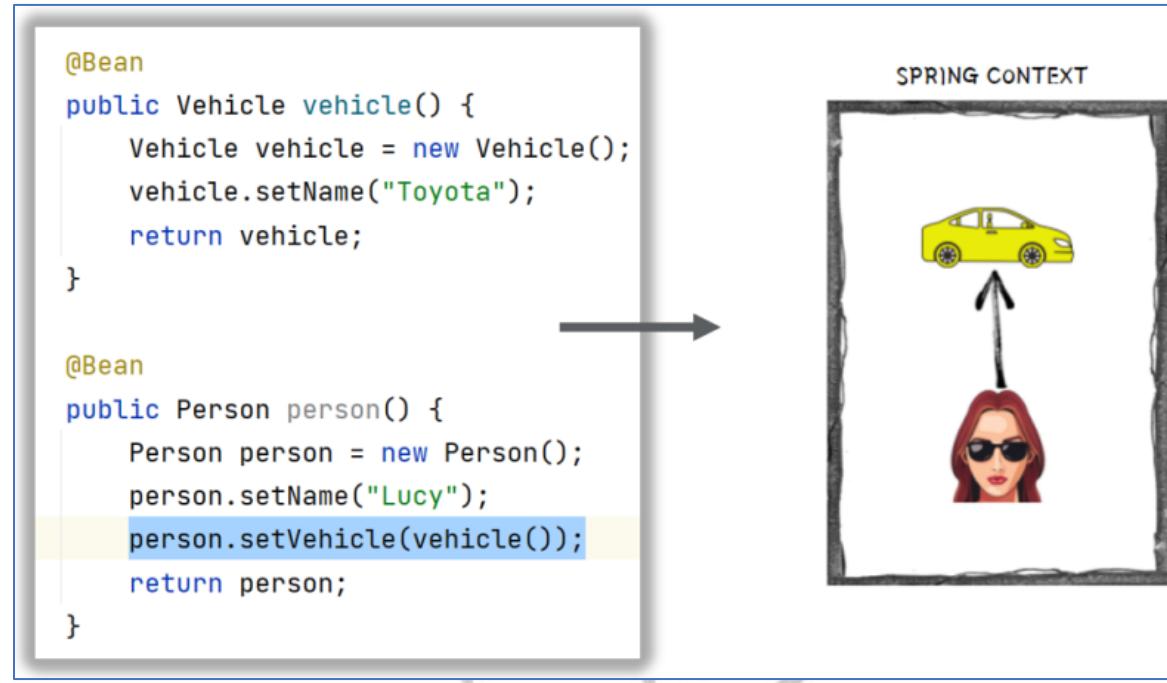


SPRING CONTEXT WITH
WIRING & DI



2. Wiring Beans using method call

Hãy xem xét một kịch bản trong đó có hai lớp java Person và Vehicle. Lớp Person có sự phụ thuộc vào Vehicle. Dựa trên đoạn mã dưới đây, chúng tôi chỉ tạo các bean bên trong Spring Context và sẽ không thực hiện nối dây. Do đó, cả hai Bean này đều có mặt trong Spring Context mà không biết về nhau.



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```

Output on Console

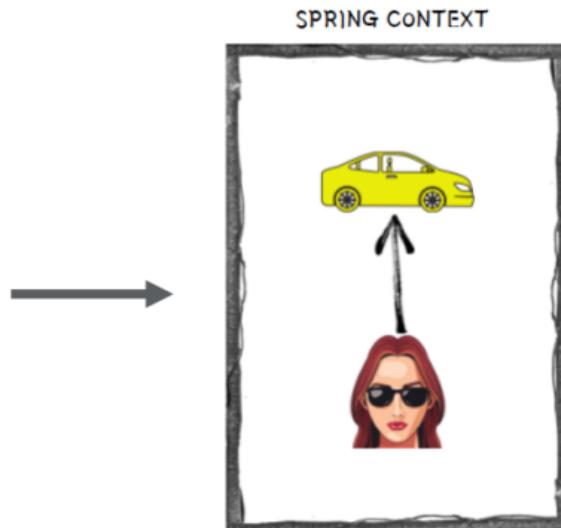
```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

3. Wiring Beans using method parameters

Ở đây trong đoạn mã dưới đây, chúng tôi đang cố gắng kết nối hoặc thiết lập mối quan hệ giữa Person và Vehicle, bằng cách gọi phương thức vehicle() từ phương thức person(). Bây giờ bên trong Sprint Context, person sở hữu vehicle. Spring sẽ đảm bảo chỉ có 1 vehicle bean được tạo và vehicle bean cũng sẽ luôn được tạo trước vì person bean phụ thuộc vào nó.

```
@Bean
public Vehicle vehicle() {
    Vehicle vehicle = new Vehicle();
    vehicle.setName("Toyota");
    return vehicle;
}

/*
@Bean
public Person person(Vehicle vehicle) {
    Person person = new Person();
    person.setName("Lucy");
    person.setVehicle(vehicle);
    return person;
}
```



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```



Output on Console

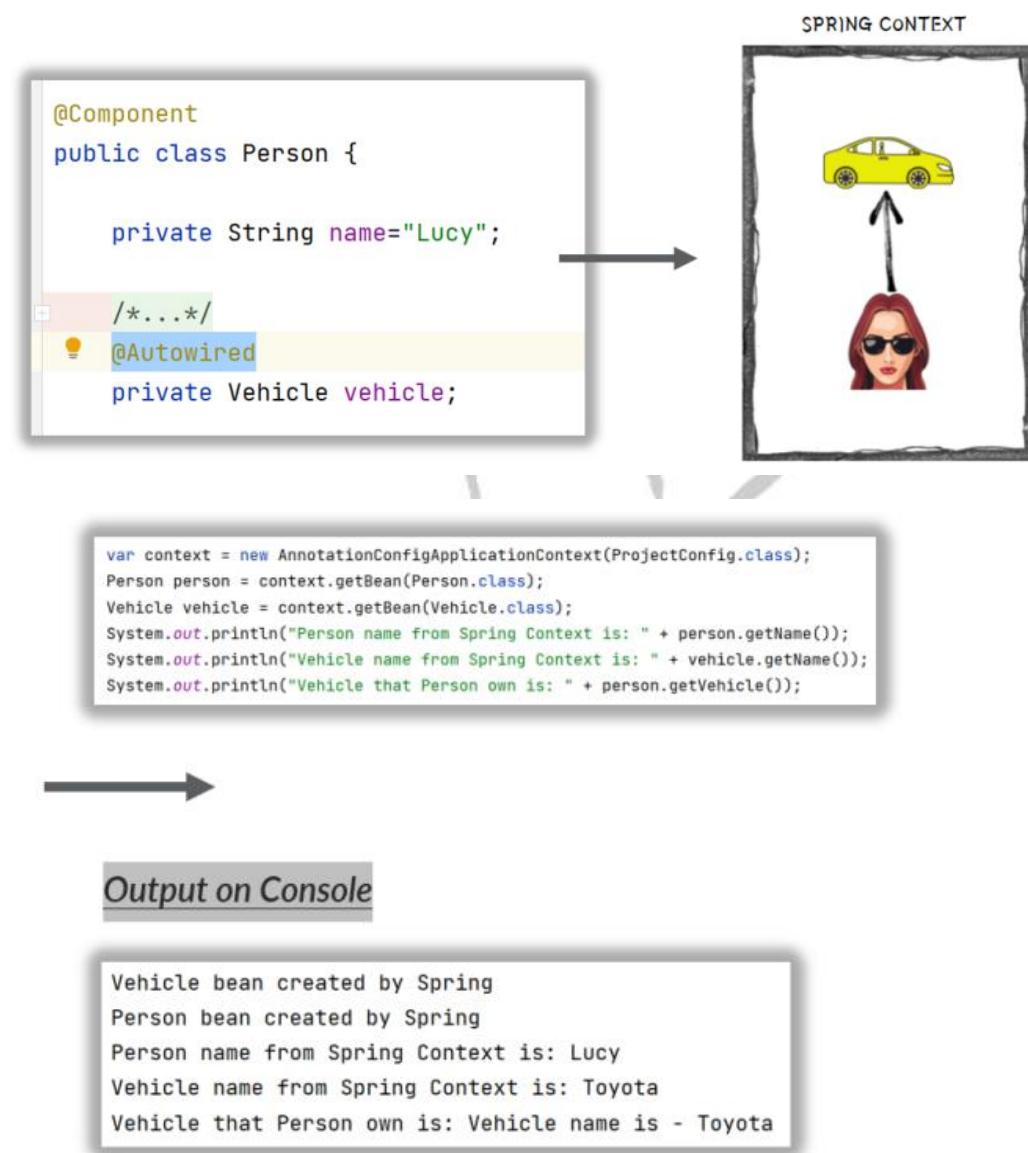
```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

4. Wiring Beans using @Autowired on class fields (example 11)

@Autowired annotation đánh dấu trên một trường, phương thức setter, hàm tạo được sử dụng để tự động kết nối các bean đang 'injecting beans'(Objects) vào thời gian chạy theo cơ chế Spring Dependency Injection.

Với đoạn mã dưới đây, Spring injects/auto-wire kết vehicle bean với person bean thông qua class field và dependency injection.

Kiểu bên dưới không được khuyến nghị cho việc sử dụng production vì chúng tôi không thể đánh dấu các trường là final.

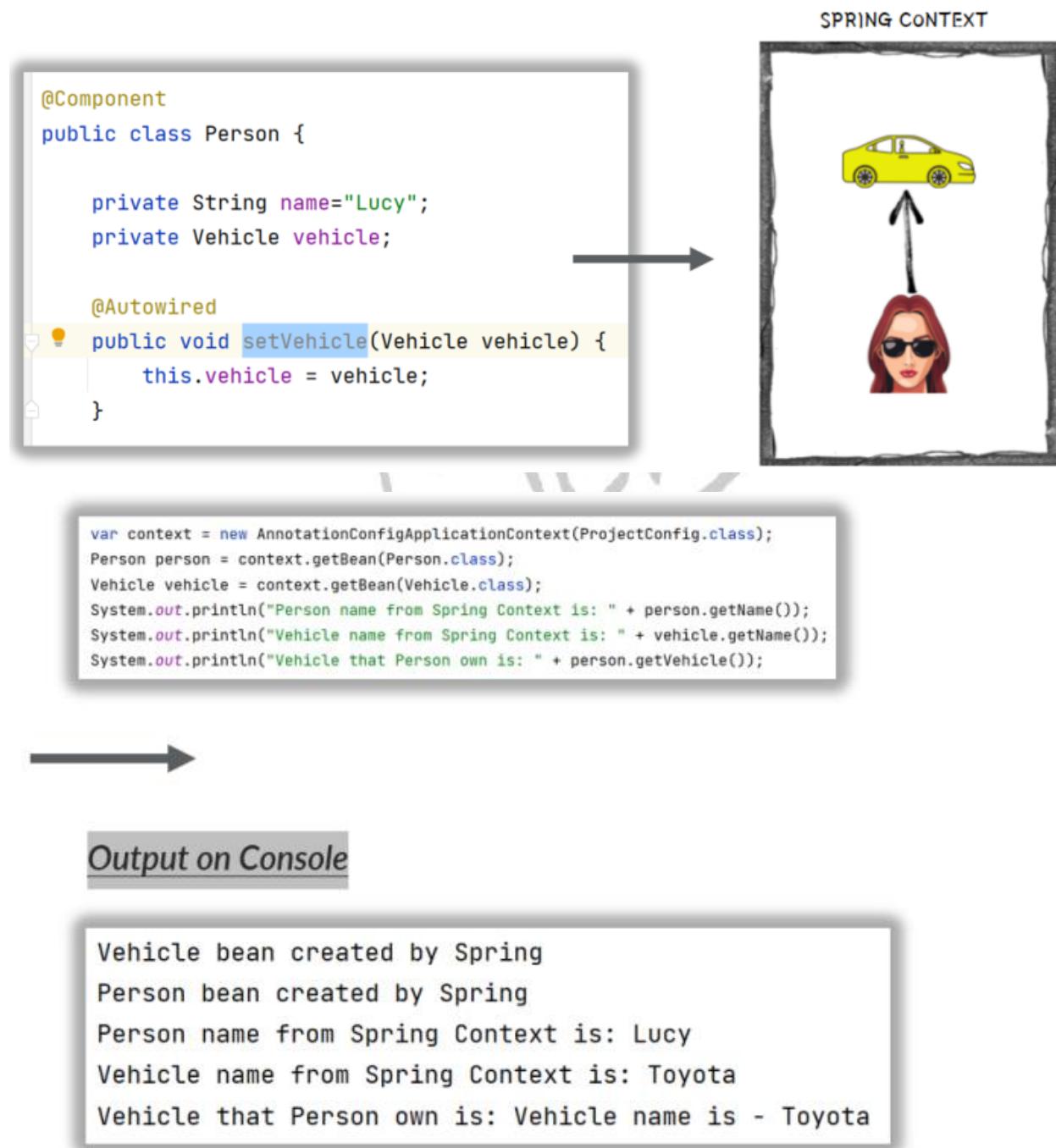


=> `@Autowired(required = false)` sẽ giúp tránh `NoSuchBeanDefinitionException` nếu bean không có sẵn trong quá trình Autowiring.

5. Wiring Beans using @Autowired on setter method

Với đoạn mã dưới đây, Spring injects/auto-wire kết nối vehicle bean với person bean thông qua phương thức setter và dependency injection.

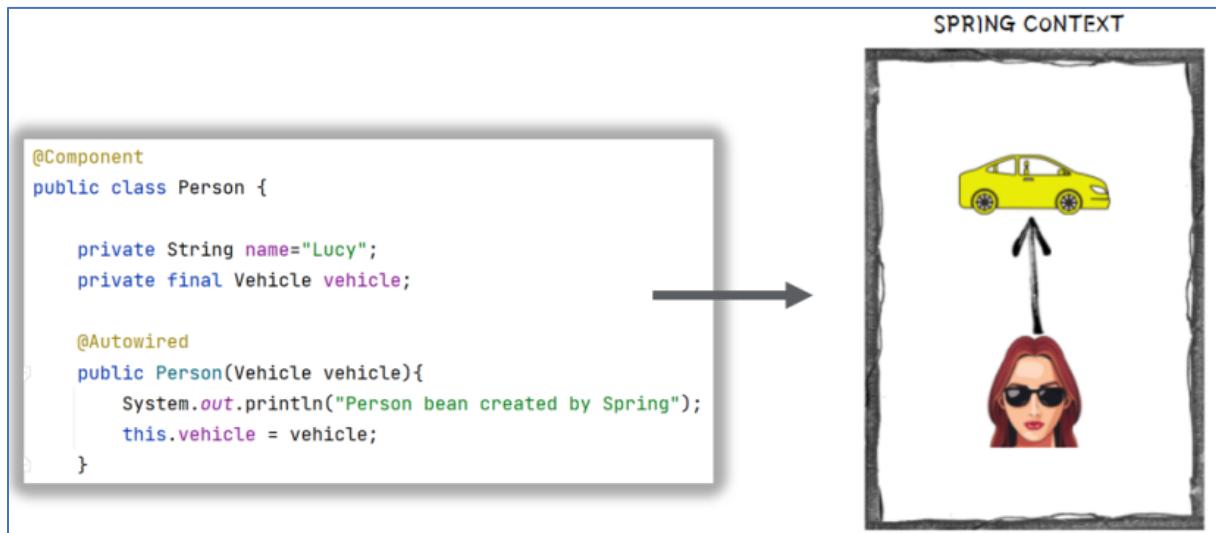
Kiểu bên dưới không được khuyến nghị cho việc sử dụng production vì chúng tôi không thể đánh dấu các trường là final và không thân thiện với người đọc.



6. Wiring Beans using @Autowired on constructor

Với đoạn mã dưới đây, Spring injects/auto-wire kết nối vehicle bean với person bean thông qua hàm constructor và dependency injection.

Từ phiên bản Spring 4.3, khi chỉ có một hàm constructor trong lớp, việc viết annotation @Autowired là tùy chọn.



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```

Output on Console

```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

7. Deep dive of Autowiring inside Spring (example12)

Theo mặc định, Spring cố gắng tự động kết nối với loại lớp. Nhưng cách tiếp cận này sẽ thất bại nếu cùng một loại lớp có nhiều bean.

Nếu Spring context có nhiều bean cùng loại như bên dưới, thì Spring sẽ cố gắng auto-wire dựa trên tên name/field mà sử dụng trong khi định cấu hình autowiring annotation.

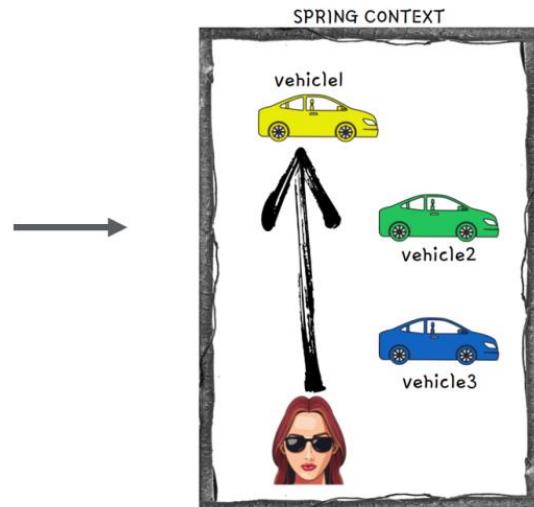
Trong trường hợp bên dưới, chúng tôi đã sử dụng 'vehicle1' làm tham số hàm constructor. Spring sẽ cố gắng auto-wire với bean có cùng tên như trong hình bên dưới.

```
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle1){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle1;
    }
}
```

STEP 1



```
public class ProjectConfig {

    @Bean
    Vehicle vehicle1() {
        var veh = new Vehicle();
        veh.setName("Audi");
        return veh;
    }

    @Bean
    Vehicle vehicle2() {
        var veh = new Vehicle();
        veh.setName("Honda");
        return veh;
    }

    @Bean
    Vehicle vehicle3() {
        var veh = new Vehicle();
        veh.setName("Ferrari");
        return veh;
    }
}
```

Nếu tên name/field mà sử dụng trong khi định cấu hình autowiring annotation không khớp với bất kỳ tên bean nào, thì Spring sẽ tìm bean đã cấu hình @Primary.

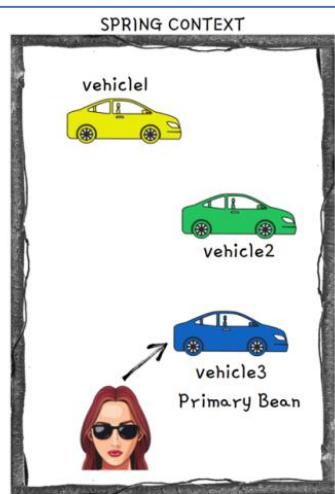
Trong trường hợp bên dưới, chúng tôi đã sử dụng 'vehicle' làm tham số hàm constructor. Spring sẽ cố gắng auto-wire với bean có cùng tên và vì nó không thể tìm thấy bean có cùng tên, nên nó sẽ tìm bean có cấu hình @Primary như trong hình bên dưới.

```
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle;
    }
}
```

STEP 2



```
public class ProjectConfig {

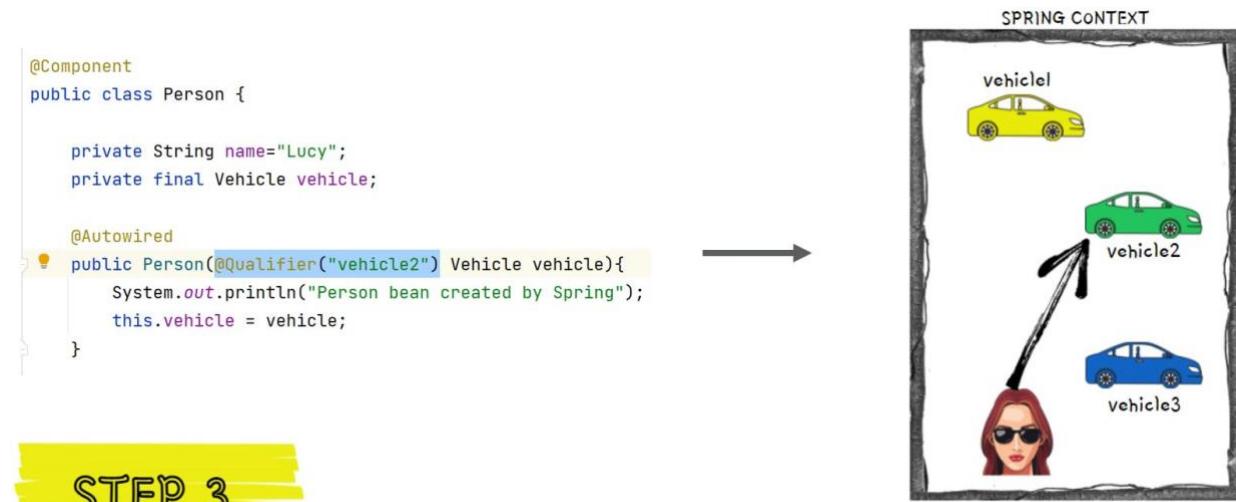
    @Bean
    @Primary
    Vehicle vehicle1() {
        var veh = new Vehicle();
        veh.setName("Audi");
        return veh;
    }

    @Bean
    Vehicle vehicle2() {
        var veh = new Vehicle();
        veh.setName("Honda");
        return veh;
    }

    @Bean
    Vehicle vehicle3() {
        var veh = new Vehicle();
        veh.setName("Ferrari");
        return veh;
    }
}
```

+ Nếu tên name/field mà sử dụng trong khi định cấu hình autowiring annotation không khớp với bất kỳ tên bean nào và ngay cả Primary bean cũng không được định cấu hình, thì Spring sẽ xem xét Annotation @Qualifier có được sử dụng với tên bean phù hợp với Spring context.

Trong kịch bản bên dưới, chúng tôi đã sử dụng 'vehicle2' với Annotation @Qualifier annotation. Spring sẽ cố gắng auto-wire với bean có cùng tên như trong hình bên dưới.



STEP 3

Chú ý nếu **@Primary** và **@Qualifier** cùng được sử dụng thì Spring sẽ chọn **@Qualifier**

8. Understanding & Avoiding Circular dependencies

Trong Spring Framework, circular dependencies (phụ thuộc vòng) xảy ra khi hai hoặc nhiều beans phụ thuộc lẫn nhau theo cách tạo ra một chuỗi vòng lặp. Điều này có thể gây ra các vấn đề và khó khăn trong quá trình tạo và quản lý các beans.

Để hiểu và tránh circular dependencies trong Spring, hãy cùng tìm hiểu một số khái niệm quan trọng:

Circular Dependencies là gì?

Circular dependencies xảy ra khi hai hoặc nhiều beans tạo thành một vòng phụ thuộc, tức là bean A phụ thuộc vào bean B, bean B phụ thuộc vào bean C, và bean C phụ thuộc lại vào bean A.

Vấn đề của Circular Dependencies:

Circular dependencies có thể gây ra các vấn đề như:

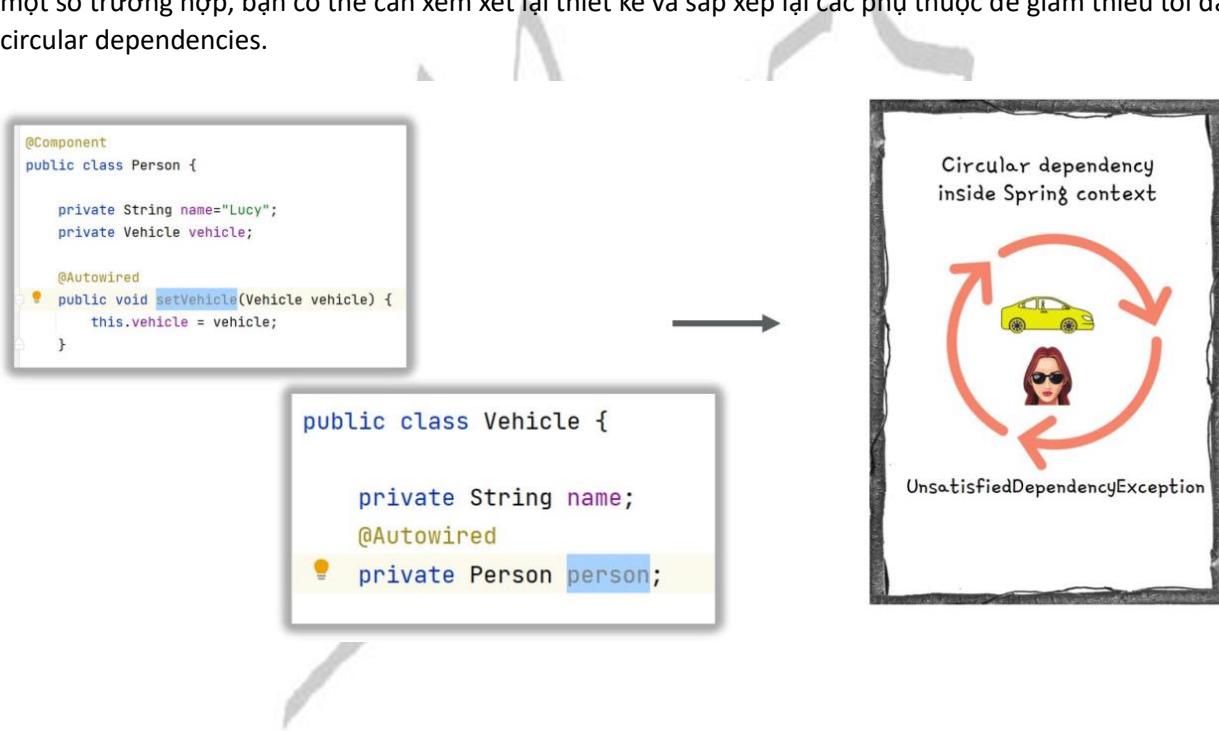
- Khó khăn trong việc khởi tạo beans: Spring không thể xác định thứ tự khởi tạo các beans trong một chuỗi phụ thuộc vòng.
- Rủi ro Deadlock: Nếu các phụ thuộc giữa các beans không được quản lý cẩn thận, có thể xảy ra deadlock khi các beans cố gắng khởi tạo và phụ thuộc vào nhau.
- Giảm khả năng kiểm tra và bảo trì: Vì các beans phụ thuộc lẫn nhau, việc kiểm tra, sửa lỗi và bảo trì có thể trở nên phức tạp hơn.

Cách tránh Circular Dependencies:

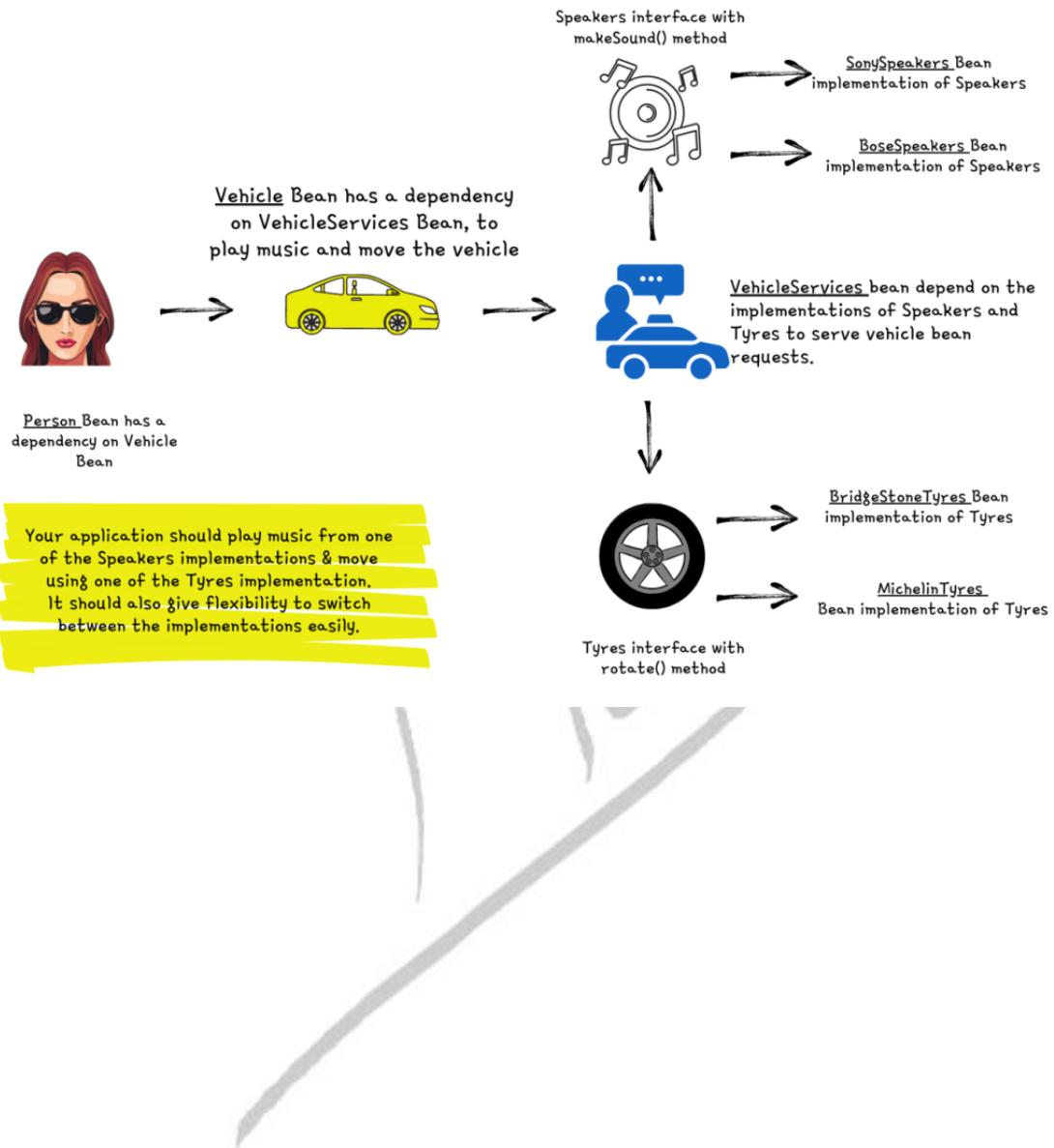
Để tránh circular dependencies trong Spring, bạn có thể áp dụng các phương pháp sau:

- Sử dụng Dependency Injection qua Constructor: Thay vì sử dụng injection qua các thuộc tính hoặc setter method, hãy sử dụng injection qua constructor. Điều này giúp xác định rõ ràng các phụ thuộc và tránh vòng phụ thuộc.
- Sử dụng Setter Injection hoặc Field Injection: Nếu bạn không thể sử dụng constructor injection, hãy sử dụng setter injection hoặc field injection. Tuy nhiên, đảm bảo rằng không có phụ thuộc vòng trong quá trình injection.
- Sử dụng Lazy Initialization: Sử dụng lazy initialization để trì hoãn việc khởi tạo các beans và tránh circular dependencies trong quá trình khởi tạo ban đầu.
- Tái cấu trúc các beans: Xem xét lại cấu trúc của các beans và xác định xem có thể tách chúng thành các beans riêng lẻ để tránh circular dependencies.

Tuy nhiên, tránh hoàn toàn circular dependencies không phải lúc nào cũng thực tế hoặc dễ dàng. Trong một số trường hợp, bạn có thể cần xem xét lại thiết kế và sắp xếp lại các phụ thuộc để giảm thiểu tối đa circular dependencies.



9. Problem Statement for Assignment related to Beans, Autowiring and DI (example13)



Section 4: Beans scope inside Spring framework

1. Introduction to Bean Scopes inside Spring

Trong Spring Framework, bean scope (phạm vi bean) là một khái niệm quan trọng để xác định thời gian tồn tại và sự sử dụng của một bean trong container. Spring hỗ trợ nhiều loại bean scopes khác nhau, mỗi loại phù hợp với các tình huống sử dụng khác nhau.

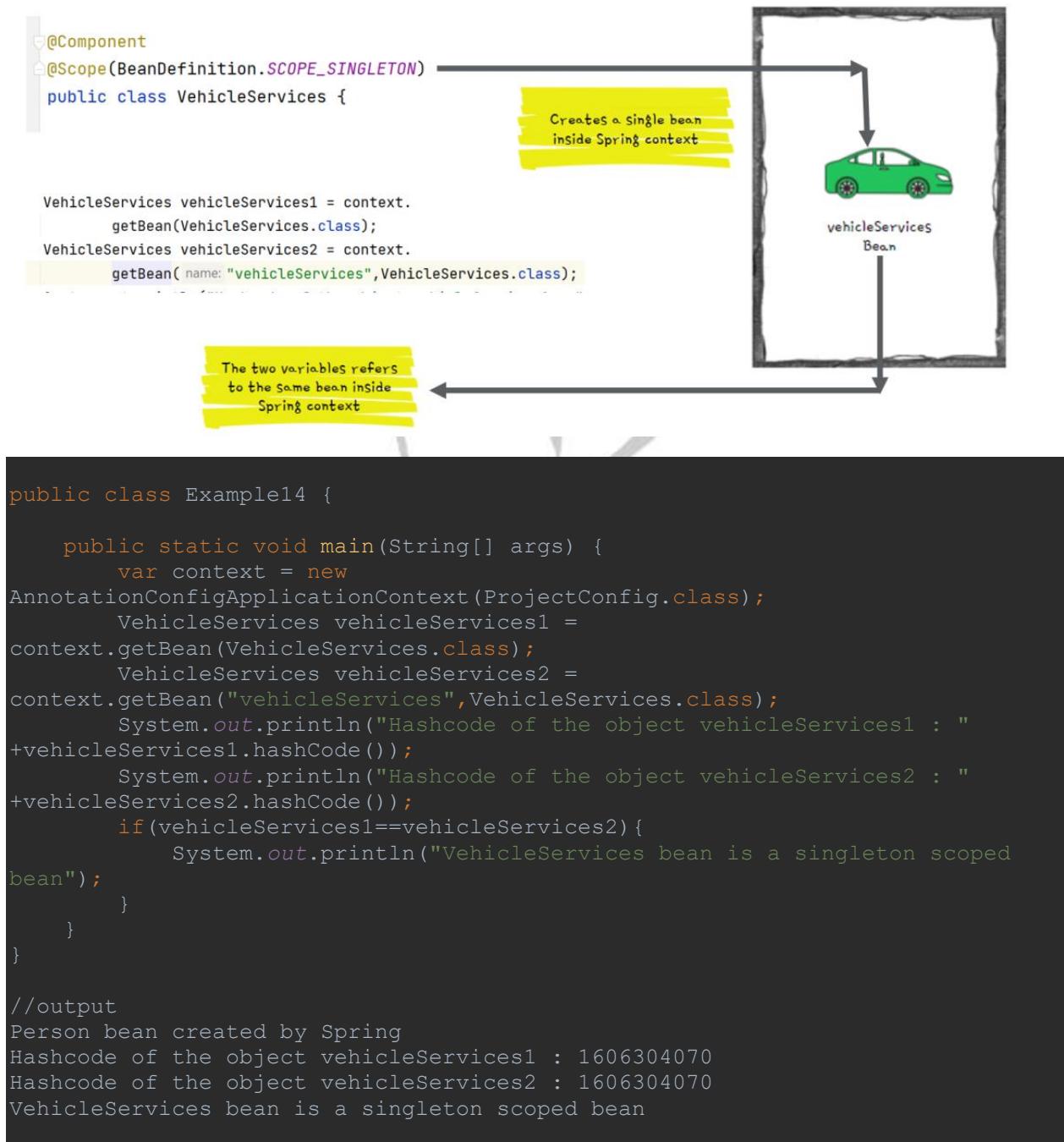
Dưới đây là một số loại bean scopes phổ biến trong Spring:

1. Singleton: Đây là mặc định scope của một bean trong Spring. Một bean với scope singleton chỉ có một phiên bản duy nhất trong toàn bộ container. Mỗi lần bean được yêu cầu, Spring sẽ cung cấp phiên bản đã tồn tại hoặc tạo mới nếu chưa có. Singleton scope đảm bảo rằng chỉ có một instance duy nhất của bean và được chia sẻ trong toàn bộ ứng dụng.
2. Prototype: Mỗi lần bean được yêu cầu, Spring sẽ tạo ra một phiên bản mới. Điều này đảm bảo rằng mỗi bean được sử dụng đều là một instance riêng biệt. Prototype scope thích hợp cho các bean có trạng thái (stateful) hoặc khi cần tạo ra nhiều phiên bản của bean.
3. Request: Mỗi bean trong scope này sẽ tồn tại trong suốt một request HTTP. Điều này có nghĩa là mỗi request sẽ có một instance riêng của bean. Scope này thường được sử dụng trong môi trường web để đảm bảo rằng các bean không chia sẻ giữa các request.
4. Session: Mỗi bean trong scope này tồn tại trong suốt một session HTTP. Điều này đảm bảo rằng các bean chỉ được chia sẻ giữa các yêu cầu của cùng một session. Scope này thích hợp cho các bean lưu trữ dữ liệu liên quan đến một phiên làm việc của người dùng trong ứng dụng web.
5. Global Session: Tương tự như session scope, nhưng được sử dụng trong môi trường portlet. Mỗi bean trong scope này tồn tại trong suốt một global session portlet.
6. Application: Bean có scope application tồn tại trong suốt vòng đời của ứng dụng. Một instance duy nhất của bean sẽ được tạo ra và được chia sẻ cho toàn bộ ứng dụng. Scope này thích hợp cho các bean không thay đổi và cần chia sẻ thông qua toàn bộ ứng dụng.
7. Websocket: Đây là scope mới được giới thiệu từ Spring 4.2, được sử dụng cho các bean liên quan đến WebSocket trong ứng dụng web.

2. Deepdive on Singleton Bean scope (example14)

Singleton là phạm vi mặc định của bean trong Spring. Trong phạm vi này, đối với một bean duy nhất, chúng tôi luôn nhận được một trường hợp tương tự khi bạn refer hoặc autowire bên trong ứng dụng của mình.

Không giống như mẫu thiết kế Singleton nơi chỉ có 1 phiên bản trong toàn bộ ứng dụng, bên trong **Singleton scope**, Spring sẽ đảm bảo chỉ có 1 phiên bản cho mỗi bean duy nhất. Ví dụ: nếu bạn có nhiều bean cùng loại, thì phạm vi Spring Singleton sẽ duy trì 1 phiên bản cho mỗi bean được khai báo cùng loại.

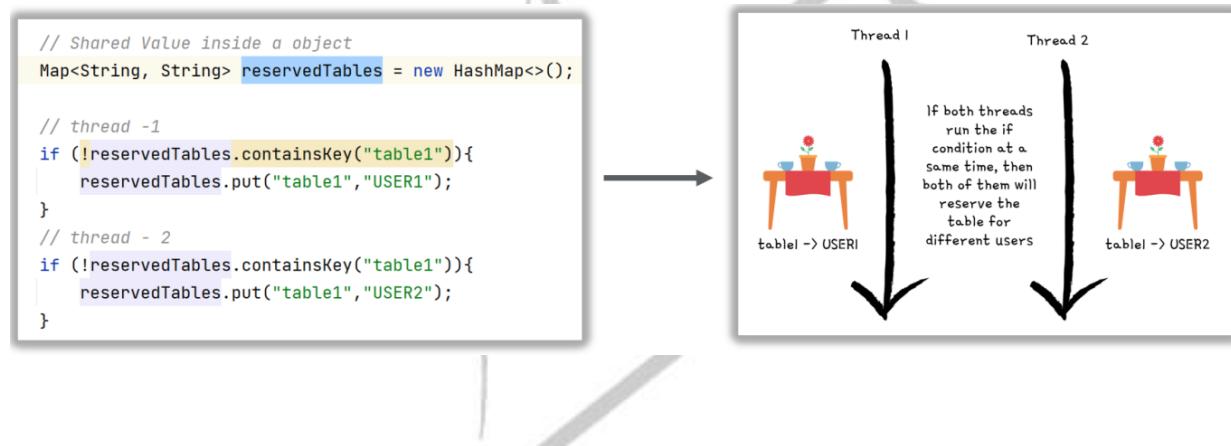


3. What is a Race Condition

Race condition (điều kiện đua) là một tình huống xảy ra trong lập trình đồng thời khi hành vi của chương trình phụ thuộc vào thứ tự hoặc thời gian thực thi của các luồng (threads) hoặc quá trình (processes) khác nhau. Nó xảy ra khi nhiều luồng hoặc quá trình truy cập vào dữ liệu hoặc tài nguyên chung cùng một lúc, và kết quả cuối cùng của chương trình phụ thuộc vào thứ tự cụ thể mà các luồng được lập lịch chạy.

Trong một race condition, việc thực thi và kết quả đúng của chương trình phụ thuộc vào việc xen kẽ (interleaving) hoặc lập lịch của các luồng, điều này có thể không đoán trước được và không xác định. Điều này có thể dẫn đến kết quả không mong muốn và sai sót, vì các luồng có thể gây ảnh hưởng lẫn nhau hoặc ghi đè lên dữ liệu của nhau.

Race condition thường xảy ra khi nhiều luồng hoặc quá trình thực hiện các hoạt động đọc-sửa-ghi (read-modify-write) trên dữ liệu chung mà không có các cơ chế đồng bộ hóa hoặc phối hợp đúng đắn. Ví dụ, nếu hai luồng cố gắng tăng một biến đếm chia sẻ cùng một lúc, giá trị cuối cùng của biến đếm có thể không chính xác do việc xen kẽ các hoạt động.



3. Singleton Beans use cases

Vì cùng một phiên bản của singleton bean sẽ được sử dụng bởi nhiều luồng bên trong ứng dụng của bạn, nên điều rất quan trọng là các bean này là bất biến.

Phạm vi này phù hợp hơn cho các bean xử lý service layer, repository layer, business logics.

1. Xây dựng các singleton bean có thể thay đổi, sẽ dẫn đến các race conditions bên trong môi trường đa luồng.
2. Có nhiều cách để tránh các race condition do singleton beans có thể thay đổi với sự trợ giúp của đồng bộ hóa.
3. Nhưng nó không được khuyến khích, vì nó mang lại nhiều vấn đề phức tạp và hiệu suất bên trong ứng dụng của bạn. Vì vậy, vui lòng không cố gắng xây dựng các loại singleton bean có thể thay đổi.

Singleton bean scope trong Spring được sử dụng để đảm bảo rằng chỉ có một phiên bản duy nhất của một bean được tạo ra và chia sẻ trong toàn bộ ứng dụng. Đây là một trong những phạm vi bean phổ biến nhất trong Spring và được sử dụng trong nhiều trường hợp khác nhau. Dưới đây là một số use case phổ biến của Singleton bean scope:

- Cấu hình ứng dụng: Singleton scope được sử dụng cho các bean chịu trách nhiệm cấu hình và cung cấp dịch vụ cấu hình cho toàn bộ ứng dụng. Ví dụ: bean chịu trách nhiệm đọc cấu hình từ tệp cấu hình và cung cấp các giá trị cấu hình cho các thành phần khác trong ứng dụng.
- Kết nối cơ sở dữ liệu: Singleton scope thường được sử dụng cho các bean quản lý kết nối cơ sở dữ liệu. Thay vì tạo ra một kết nối mới cho mỗi yêu cầu, một bean Singleton có thể duy trì một kết nối cơ sở dữ liệu duy nhất và chia sẻ nó giữa các thành phần trong ứng dụng.
- Cache: Singleton scope cũng thích hợp cho các bean quản lý cache dữ liệu. Bean Singleton có thể duy trì một bộ nhớ cache và cung cấp dữ liệu cached cho các thành phần khác trong ứng dụng.
- Dịch vụ: Singleton scope thường được sử dụng cho các bean đại diện cho các dịch vụ cung cấp chức năng cho toàn bộ ứng dụng. Ví dụ: bean quản lý email service, logging service, authentication service, etc.
- Thành phần chia sẻ: Singleton scope được sử dụng cho các thành phần được chia sẻ trong toàn bộ ứng dụng. Ví dụ: bean chịu trách nhiệm quản lý danh sách người dùng đăng nhập, bean chịu trách nhiệm quản lý danh sách các phiên làm việc hiện tại của người dùng.

Lưu ý rằng việc sử dụng Singleton scope cần cẩn thận để đảm bảo tính nhất quán và an toàn khi truy cập đồng thời vào dữ liệu chia sẻ. Đồng thời, cần chú ý xử lý các vấn đề liên quan đến đồng bộ hóa và thread safety khi sử dụng Singleton bean trong môi trường đa luồng.

4. Deepdive of Eager and Lazy instantiation of Singleton scope (example15)

Trong Spring, Singleton là một trong những phạm vi (scope) bean mặc định, có nghĩa là một instance của bean sẽ được tạo ra duy nhất và được chia sẻ trong toàn bộ ứng dụng. Tuy nhiên, có hai cách để thực hiện việc khởi tạo Singleton bean: eager (sẵn sàng) và lazy (lười biếng) instantiation.

Theo mặc định, Spring sẽ khởi tạo Singleton bean: eager (sẵn sàng) trong quá trình khởi động ứng dụng. Điều này được gọi là Eager instantiation.

a) Eager Instantiation (Khởi tạo sẵn sàng):

Trong eager instantiation, Singleton bean được tạo ra ngay khi ApplicationContext khởi động hoặc tạo ra một instance của bean nằm trong Singleton scope. Điều này có nghĩa là bean sẽ được tạo ra trước khi nó được yêu cầu lần đầu tiên và sẵn sàng sử dụng ngay từ đầu. Khi eager instantiation được sử dụng, tất cả các Singleton bean sẽ được tạo ra cùng với ApplicationContext và sẵn sàng để sử dụng trong toàn bộ vòng đời của ứng dụng.

b) Lazy Instantiation (Khởi tạo lười biếng):

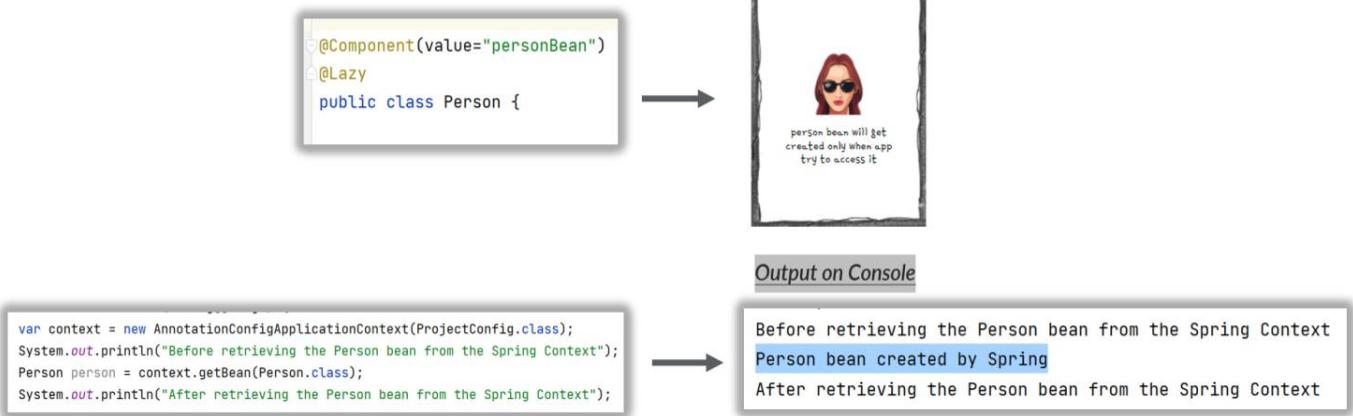
Trái ngược với eager instantiation, trong lazy instantiation, Singleton bean không được tạo ra ngay khi ApplicationContext khởi động. Thay vào đó, bean chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên. Khi một bean Singleton được yêu cầu, nó sẽ được tạo ra và lưu trữ để sử dụng trong các yêu cầu tiếp theo. Lazy instantiation giúp giảm tải khởi động ban đầu của ứng dụng và chỉ tạo ra các bean cần thiết.

Sự lựa chọn giữa eager và lazy instantiation phụ thuộc vào yêu cầu của ứng dụng và tài nguyên hệ thống. Một số điểm cần lưu ý khi quyết định sử dụng eager hoặc lazy instantiation cho Singleton bean:

- Eager instantiation thích hợp trong trường hợp bean có khối lượng công việc nặng hoặc cần sử dụng ngay từ đầu của ứng dụng.
- Lazy instantiation thích hợp khi tài nguyên hệ thống quan trọng và cần được tiết kiệm, hoặc khi bean có thể không cần thiết trong mọi trường hợp sử dụng.

Eager Instantiation	Lazy Instantiation
<ul style="list-style-type: none"> • Đây là hành vi mặc định bên trong Spring framework • Singleton bean sẽ được tạo trong quá trình khởi động ứng dụng • Máy chủ sẽ không khởi động nếu bean không thể tạo do bất kỳ ngoại lệ phụ thuộc nào • Spring context sẽ chiếm nhiều bộ nhớ nếu cố gắng sử dụng eager cho tất cả các bean bên trong ứng dụng • Eager có thể được theo dõi cho tất cả các loại bean được yêu cầu rất phổ biến trong một ứng dụng 	<ul style="list-style-type: none"> • Đây không phải là hành vi mặc định và cần định cấu hình rõ ràng bằng cách sử dụng <code>@Lazy</code> • Singleton bean sẽ được tạo khi ứng dụng cố gắng giới thiệu bean lần đầu tiên • Ứng dụng sẽ đưa ra exception runtime nếu việc tạo bean không thành công do bất kỳ ngoại lệ phụ thuộc nào • Hiệu suất sẽ bị ảnh hưởng nếu cố gắng sử dụng lazy cho tất cả các bean bên trong ứng dụng • Lazy có thể theo dõi đối với các bean được sử dụng trong một tình huống rất xa bên trong một ứng dụng

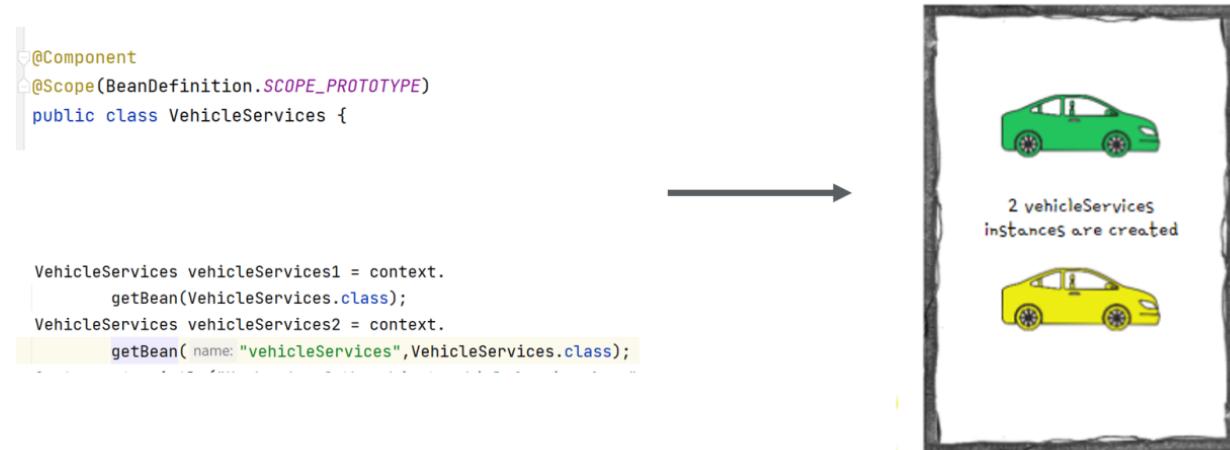
Sample demo of Lazy instantiation



5. Deepdive of Prototype Bean scope (example16)

Với prototype scope, mỗi khi yêu cầu tham chiếu một bean, Spring sẽ tạo một thể hiện đối tượng mới và cung cấp tương tự.

Prototype scope hiếm khi được sử dụng bên trong các ứng dụng và chúng tôi chỉ có thể sử dụng phạm vi này trong các tình huống mà bean của bạn sẽ thường xuyên thay đổi trạng thái của dữ liệu, điều này sẽ dẫn đến các điều kiện tương tranh trong môi trường đa luồng. Sử dụng prototype scope sẽ không tạo ra bất kỳ race condition nào.



6. Singleton Beans Vs Prototype Beans

Singleton Beans	Prototype Beans
<ul style="list-style-type: none">Đây là phạm vi mặc định bên trong Spring frameworkCùng một thể hiện đối tượng sẽ được trả về mỗi khi giới thiệu một bean bên trong mãChúng ta có thể cấu hình để tạo các bean trong quá trình khởi động hoặc khi tham chiếu lần đầuCác đối tượng bất biến có thể không hoạt động đối với phạm vi SingletonScope được sử dụng phổ biến nhất	<ul style="list-style-type: none">Cần cấu hình rõ ràng bằng cách sử dụng @Scope(SCOPE_PROTOTYPE)Đối tượng mới sẽ được trả về mỗi giới thiệu một bean bên trong mãSpring luôn tạo đối tượng mới khi cố gắng tham chiếu bean. Không thể eager instantiationCác đối tượng có thể thay đổi có thể không hoạt động đối với phạm vi nguyên mẫuPhạm vi rất hiếm khi được sử dụng

Section 5: Aspect Oriented Programming (AOP) inside Spring framework

1. Introduction to Aspect Oriented Programming (AOP)

Một **Aspect** (khía cạnh) chỉ đơn giản là một đoạn mã mà Spring framework thực thi khi bạn gọi các phương thức cụ thể bên trong ứng dụng của mình.

Spring AOP cho phép Aspect-Oriented Programming (Lập trình hướng khía cạnh) trong các spring applications. Trong AOP, các khía cạnh cho phép mô đun hóa các mối quan tâm như transaction management, logging hay security xuyên suốt nhiều loại và đối tượng (thường được gọi là mối quan tâm xuyên suốt)

- AOP cung cấp cách để tự động thêm mối quan tâm xuyên suốt trước, sau hoặc xung quanh(before, after, around) logic thực tế bằng cách sử dụng các cấu hình đơn giản.
- AOP giúp phân tách và duy trì nhiều mã liên quan đến logic non-business như logging, auditing, security, transaction management.
- AOP là một mô hình lập trình nhằm mục đích tăng tính mô đun hóa bằng cách cho phép tách biệt các mối quan tâm xuyên suốt. Nó thực hiện điều này bằng cách thêm hành vi bổ sung vào mã hiện có mà không sửa đổi mã

2. AOP Jargons – Thuật ngữ AOP

Khi định nghĩa một Aspect hoặc thực hiện cấu hình, cần tuân theo WWW (3Ws)

- WHAT -> Aspect
- WHEN -> Advice
- WHICH -> Pointcut

WHAT – Cái gì Mã hoặc logic nào mà chúng tôi muốn Spring thực thi khi bạn gọi một phương thức cụ thể. Điều này được gọi là Aspect (Khía cạnh).

WHEN – Khi nào Spring cần thực thi Aspect đã cho. Ví dụ: trước hoặc sau khi gọi phương thức. Điều này được gọi là Advice.

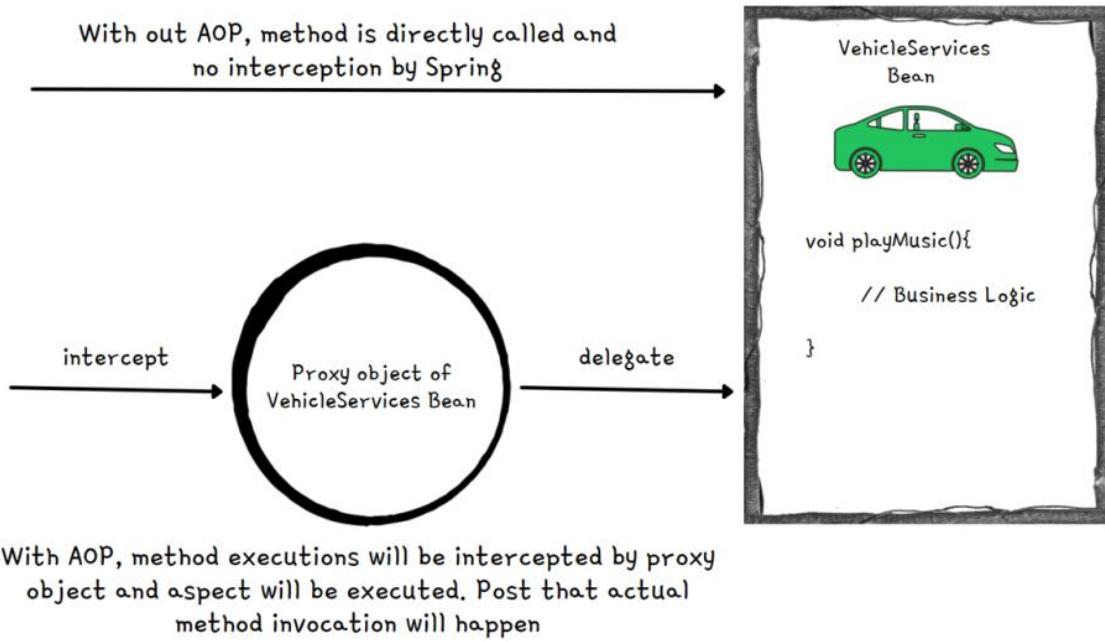
WHICH - Cài mà Phương thức bên trong Ứng dụng mà khung đó cần chặn và thực thi Aspect đã cho. Điều này được gọi là Pointcut.

- Join point xác định sự kiện kích hoạt việc thực thi một aspect. Trong Spring, sự kiện này luôn là một cuộc gọi phương thức.
- Target object là bean khai báo phương thức/pointcut bị chặn bởi một khía cạnh.

3. Weaving inside AOP

Khi triển khai AOP bên trong application của mình bằng Spring framework, nó sẽ chặn(intercept) từng lệnh gọi phương thức và áp dụng logic được xác định trong Aspect.

Nhưng làm thế nào để điều này hoạt động? Spring thực hiện điều này với sự trợ giúp của đối tượng proxy. Vì vậy, chúng tôi cố gắng gọi một phương thức bên trong một bean, Spring thay vì trực tiếp đưa ra tham chiếu của bean thay vào đó, nó sẽ đưa ra một đối tượng proxy sẽ quản lý từng cuộc gọi đến một phương thức và áp dụng logic khía cạnh. Quá trình này được gọi là **Weaving**



4. Type of Advices inside AOP

Một số loại Advice phổ biến trong AOP bao gồm:

@Before: Advice này chạy trước khi thực thi điểm liên kết và có thể được sử dụng để thực hiện một số cài đặt hoặc xác thực bổ sung trước khi thực hiện phương thức thực sự.

@After: Advice này chạy sau khi thực thi điểm liên kết, bất kể phương thức thực thi có thành công hay không. Advice sau có thể được sử dụng để thực hiện một số hoạt động dọn dẹp hoặc ghi nhật ký.

@Around: Advice này chạy quanh thực thi điểm liên kết và có thể sửa đổi hành vi của phương thức trước và sau khi thực thi. @Around là loại lời khuyên mạnh nhất và có thể được sử dụng để triển khai bộ nhớ cache, retry và các tính năng nâng cao khác.

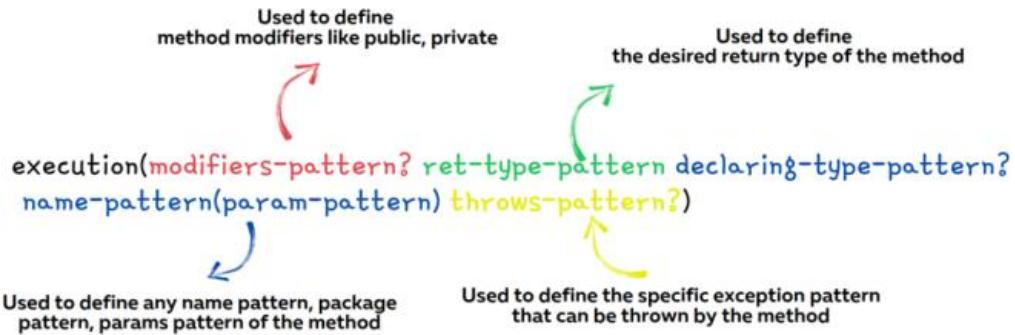
@AfterReturning: Advice này chạy sau khi thực thi thành công của điểm liên kết và có thể được sử dụng để thực hiện một số xử lý bổ sung trên giá trị trả về.

@AfterThrowing: Advice này chạy sau khi điểm liên kết ném ra một ngoại lệ và có thể được sử dụng để thực hiện một số xử lý lỗi hoặc ghi nhật ký.

5. Configuring Advices inside AOP

Execution expression là một phương pháp sử dụng trong Aspect-Oriented Programming (AOP) để xác định các điểm cắt (join points) trong mã ứng dụng. Nó cho phép bạn chỉ định các điểm cắt cụ thể mà các khía cạnh (aspects) sẽ được áp dụng.

Trong execution expression, bạn sử dụng biểu thức execution để xác định phương thức hoặc phương thức của một lớp cụ thể mà bạn muốn áp dụng advice. Cú pháp của execution expression thường được sử dụng như sau:



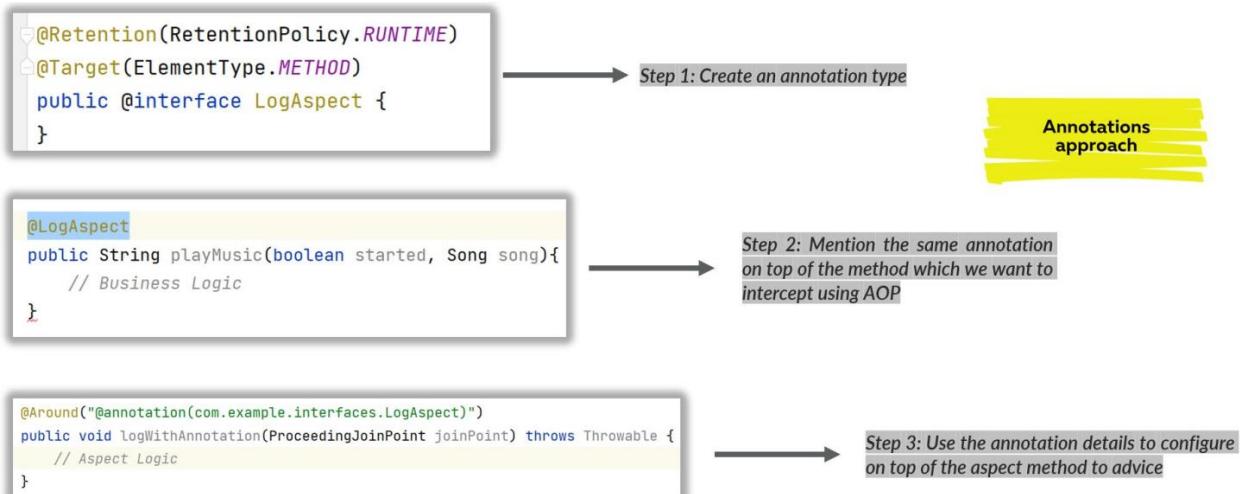
Ở đây, một số thành phần quan trọng của execution expression bao gồm:

- modifiers-pattern: Xác định các phạm vi truy cập của phương thức (ví dụ: public, protected, private).
- return-type-pattern: Xác định kiểu trả về của phương thức.
- declaring-type-pattern: Xác định lớp chứa phương thức.
- method-name-pattern: Xác định tên của phương thức.
- param-pattern: Xác định kiểu và số lượng tham số của phương thức.
- throws-pattern: Xác định các ngoại lệ mà phương thức có thể ném ra.

```
@Configuration  
@ComponentScan(basePackages = {"com.example.implementation",  
    "com.example.services", "com.example.aspects"})  
@EnableAspectJAutoProxy  
public class ProjectConfig {  
}
```

```
@Aspect  
@Component  
public class LoggerAspect {  
  
    @Around("execution(* com.example.services.*.*(..))")  
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {  
        // Aspect Logic  
    }  
}
```

Ngoài ra, có thể sử dụng kiểu Annotation để định cấu hình Advices bên trong AOP. Dưới đây là ba bước mà chúng tôi làm theo cho cùng,



6. Configuring @Around advice (example17)

@Around với annotation

Để cấu hình @Around advice trong Spring, bạn cần thực hiện các bước sau:

Bước 1: Annotation Configuration:

```
@Configuration
@ComponentScan(basePackages = {"com.example.implementation",
    "com.example.services", "com.example.aspects"})
@EnableAspectJAutoProxy
public class ProjectConfig { }
```

Bước 2: Tạo một annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LogAspect { }
```

Bước 3: Tạo một Aspect: Tạo một lớp Aspect chứa @Around advice và các thành phần AOP khác. Bạn có thể sử dụng các annotation như @Aspect để đánh dấu lớp là một Aspect và @Around để đánh dấu phương thức là advice loại @Around.

```
@Aspect
@Component
public class LoggerAspect {

    private Logger logger = Logger.getLogger(LoggerAspect.class.getName());

    @Around("@annotation(com.example.interfaces.LogAspect)")
    public void logWithAnnotation(ProceedingJoinPoint joinPoint) throws
Throwable {
        logger.info(joinPoint.toString() + " method execution start");
    }
}
```

Bước 4: Tạo một Service và sử dụng annotation

```
@Component
public class VehicleServices {

    @LogAspect
    public String playMusic(){
        return "";
    }
}
```

=> Chạy chương trình

```
public class Example17 {

    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(ProjectConfig.class);
        var vehicleServices = context.getBean(VehicleServices.class);
        System.out.println(vehicleServices.playMusic());

    }
}

// Output
Jun 28, 2023 9:41:22 PM com.example.aspects.LoggerAspect logWithAnnotation
INFO: execution(String com.example.services.VehicleServices.playMusic())
method execution start
null
```

@Around annotation áp dụng cho tất cả các phương thức trong tất cả các lớp

```
@Aspect
@Component
@Order(2)
public class LoggerAspect {

    private Logger logger = Logger.getLogger(LoggerAspect.class.getName());

    @Around("execution(* com.example.services.*.*(..))")
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info(joinPoint.getSignature().toString() + " method execution start");
        Instant start = Instant.now();
        joinPoint.proceed();
        Instant finish = Instant.now();
        long timeElapsed = Duration.between(start, finish).toMillis();
        logger.info("Time took to execute the method : "+timeElapsed);
        logger.info(joinPoint.getSignature().toString() + " method execution end");
    }
}
```

7. Configuring @Before advice

```
@Aspect
@Component
@Order(1)
public class VehicleStartCheckAspect {

    private Logger logger =
Logger.getLogger(VehicleStartCheckAspect.class.getName());

    @Before("execution(* com.example.services.*.*(..)) &&
args(vehicleStarted,..)")
    public void checkVehicleStarted(JoinPoint joinPoint, boolean
vehicleStarted) throws Throwable {
        if(!vehicleStarted){
            throw new RuntimeException("Vehicle not started");
        }
    }
}
```

Đây là một ví dụ về cách sử dụng advice @Before trong AOP để kiểm tra tham số của phương thức được gọi trước khi nó được thực thi. Nó được áp dụng cho tất cả các phương thức trong tất cả các lớp trong gói com.example.services bằng cách sử dụng pointcut expression "execution(* com.example.services..(..))". Ngoài ra, advice này sử dụng một tham số boolean được đặt tên là vehicleStarted và kiểm tra xem giá trị của tham số này có là true hay không bằng cách sử dụng args(vehicleStarted,..).

Cụ thể, @Before là một loại advice trong AOP được sử dụng để thực thi code trước khi phương thức được gọi. Trong trường hợp này, @Before được kết hợp với pointcut expression "execution(* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức

trong tất cả các lớp trong gói com.example.services. Điều kiện kiểm tra giá trị của tham số vehicleStarted được thêm vào advice bằng cách sử dụng args(vehicleStarted,..), nghĩa là chỉ áp dụng cho các phương thức có tham số đầu tiên là vehicleStarted.

Trong ví dụ này, nếu giá trị của tham số vehicleStarted là false, một ngoại lệ RuntimeException sẽ được ném ra, ngăn chặn phương thức được gọi tiếp tục thực thi và thông báo rằng "Vehicle not started". Nếu giá trị của tham số vehicleStarted là true, phương thức được gọi sẽ tiếp tục thực thi. Đối tượng JoinPoint được truyền vào advice để cho phép advice truy cập thông tin về phương thức và tham số của nó.

8. Configuring @AfterThrowing and @AfterReturning advices

```
@AfterThrowing(value = "execution(* com.example.services.*.*(..))", throwing = "ex")
public void logException(JoinPoint joinPoint, Exception ex) {
    logger.log(Level.SEVERE,joinPoint.getSignature()+" An exception thrown with the help of" +
               " @AfterThrowing which happened due to : "+ex.getMessage());
}

@AfterReturning(value = "execution(* com.example.services.*.*(..))", returning = "retval")
public void logStatus(JoinPoint joinPoint, Object retval) {
    logger.log(Level.INFO,joinPoint.getSignature()+" Method successfully processed with the status " +
               retval.toString());
}
```

Đây là một ví dụ về cách sử dụng advice @AfterThrowing và @AfterReturning trong AOP để ghi nhật ký thông tin khi một phương thức trong gói com.example.services ném ra một ngoại lệ hoặc trả về giá trị thành công.

Cụ thể, @AfterThrowing là một loại advice trong AOP được sử dụng để thực thi code sau khi phương thức ném ra ngoại lệ. Trong trường hợp này, @AfterThrowing được kết hợp với pointcut expression "execution(* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức trong tất cả các lớp trong gói com.example.services. Tham số throwing được sử dụng để truyền đối tượng ngoại lệ được ném ra.

Trong ví dụ này, khi một phương thức ném ra một ngoại lệ, advice sẽ được gọi, và đối tượng ngoại lệ được truyền vào advice. Advice này sử dụng Logger để ghi nhật ký thông tin về ngoại lệ đã xảy ra, bao gồm cả tên phương thức và thông tin về ngoại lệ đó, bằng cách sử dụng phương thức getMessage() để lấy thông tin chi tiết về ngoại lệ.

@AfterReturning là một loại advice trong AOP được sử dụng để thực thi code sau khi phương thức đã trả về giá trị thành công. Trong trường hợp này, @AfterReturning được kết hợp với pointcut expression "execution(* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức trong tất cả các lớp trong gói com.example.services. Tham số returning được sử dụng để truyền đối tượng giá trị trả về.

Trong ví dụ này, khi một phương thức trả về giá trị thành công, advice sẽ được gọi, và đối tượng giá trị trả về được truyền vào advice. Advice này sử dụng Logger để ghi nhật ký thông tin về tên phương thức và giá trị trả về, bằng cách sử dụng phương thức toString() để lấy thông tin chi tiết về giá trị trả về.

Section 6: Processing Query Params & Path Variables inside Spring

1. Accepting Query Params using @RequestParam annotation(example22)

Trong Spring, annotation @RequestParam được sử dụng để ánh xạ tham số truy vấn (query parameters) hoặc form data.

Ví dụ: nếu muốn nhận giá trị tham số từ URL được yêu cầu HTTP GET thì có thể sử dụng Annotation @RequestParam như trong ví dụ bên dưới.

```
http://localhost:8080/holidays?festival=true&federal=true
```

```
@GetMapping("/holidays")
public String displayHolidays(@RequestParam(required = false) boolean festival,
                             @RequestParam(required = false) boolean federal) {
    // Business Logic
    return "holidays.html";
}
```

Annotation @RequestParam hỗ trợ các thuộc tính như name, required, value, defaultValue. Chúng tôi có thể sử dụng chúng trong ứng dụng của mình dựa trên các yêu cầu.

- value (hoặc name): Xác định tên của tham số truy vấn. Ví dụ: @RequestParam("paramName").
- required: Xác định xem tham số truy vấn có bắt buộc hay không. Mặc định là true. Nếu đặt thành false, và tham số truy vấn không được cung cấp, Spring sẽ gán giá trị null cho tham số đó.
- defaultValue: Xác định giá trị mặc định cho tham số truy vấn khi không được cung cấp. Ví dụ: @RequestParam(value = "paramName", defaultValue = "default").

2. Accepting Query Params using @RequestParam annotation

Annotation @PathVariable được sử dụng để trích xuất giá trị từ URI. Nó phù hợp nhất cho dịch vụ web RESTful nơi URL chứa một số giá trị. Spring MVC cho phép sử dụng nhiều annotation @PathVariable trong cùng một phương thức.

```
http://localhost:8080/holidays/all
http://localhost:8080/holidays/federal
http://localhost:8080/holidays/festival
```

```
@GetMapping("/holidays/{display}")
public String displayHolidays(@PathVariable String display) {
    //Business Logic
    return "holidays.html";
}
```

=> Nó cũng hỗ trợ các attribute như name, value, required giống @RequestParam

Section 7: Validating the input using Java Bean & Hibernate Validators

1. Introduction to Java Bean Validations

+ Bean Validation (<https://beanvalidation.org/>) là tiêu chuẩn để triển khai validation trong hệ sinh thái Java. Nó được tích hợp tốt với Spring và Spring Boot.

+ Dưới đây là maven dependency mà có thể thêm vào để triển khai xác thực Bean trong bất kỳ dự án Spring/SpringBoot nào

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- **Bean Validations** hoạt động bằng cách xác định các ràng buộc đối với các trường của một lớp bằng cách Annotation chúng bằng các annotation nhất định
- chúng ta có thể đặt annotation `@Valid` trên các trường và tham số phương thức để nói với Spring rằng muốn một tham số phương thức hoặc trường được validate.
- Dưới đây là các package quan trọng nơi có thể xác định các annotation liên quan đến validate.
`jakarta.validation.constraints.*`
`org.hibernate.validator.constraints.*`

Important Validation Annotations

<code>jakarta.validation.constraints.*</code>	<code>org.hibernate.validator.constraints.*</code>
<ul style="list-style-type: none">✓ <code>@Digits</code>✓ <code>@Email</code>✓ <code>@Max</code>✓ <code>@Min</code>✓ <code>@NotBlank</code>✓ <code>@NotEmpty</code>✓ <code>@NotNull</code>✓ <code>@Pattern</code>✓ <code>@Size</code>	<ul style="list-style-type: none">✓ <code>@CreditCardNumber</code>✓ <code>@Length</code>✓ <code>@Currency</code>✓ <code>@Range</code>✓ <code>@URL</code>✓ <code>@UniqueElements</code>✓ <code>@EAN</code>✓ <code>@ISBN</code>

2. Adding Bean Validation annotations inside Contact POJO class(example24)

Khi bạn muốn thực hiện kiểm tra hợp lệ dữ liệu trên một lớp POJO trong Spring, bạn có thể sử dụng các annotation từ Java Bean Validation API. Dưới đây là ví dụ về việc thêm các annotation kiểm tra hợp lệ vào một lớp POJO Contact:

```
@Data
public class Contact {
    private String name;
    private String mobileNum;
    private String email;
    private String subject;
    private String message;
}
```

Trong ví dụ trên, sử dụng các annotation từ Java Bean Validation API để thực hiện kiểm tra hợp lệ trên các trường của lớp Contact:

- `@NotNull`: Kiểm tra xem một trường đã cho có phải là null hay không nhưng cho phép các giá trị trống & phần tử bằng 0 bên trong các bộ sưu tập.
- `@NotEmpty`: Kiểm tra xem một trường đã cho có phải là null hay không và kích thước/độ dài của nó có lớn hơn 0 hay không.
- `@NotBlank`: Kiểm tra xem một trường đã cho có phải là null không và độ dài đã cắt có lớn hơn 0 hay không.

Khi bạn sử dụng các annotation kiểm tra hợp lệ như trên, bạn cần chắc chắn rằng bạn đã cấu hình một Bean Validator trong ứng dụng Spring của mình để kiểm tra và xử lý các lỗi kiểm tra hợp lệ.

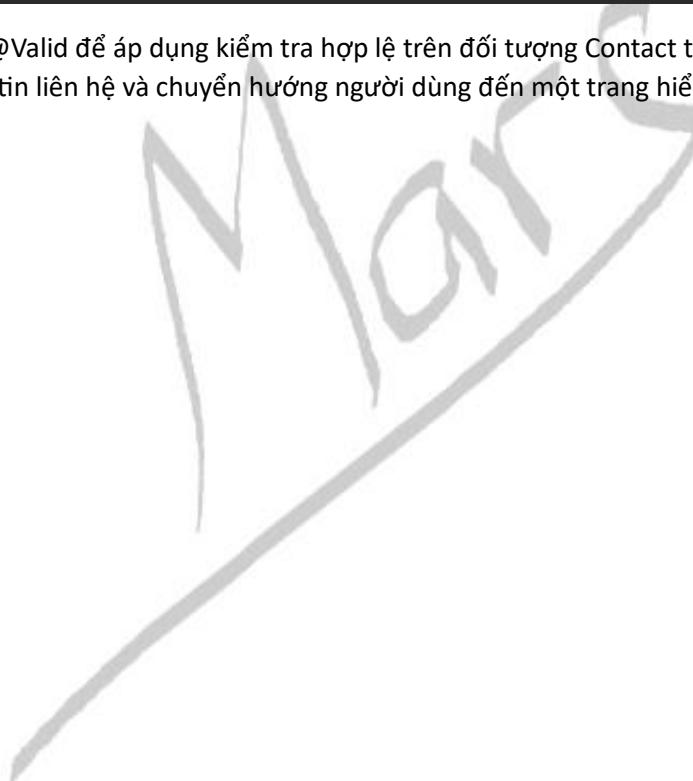
3. Adding Bean Validation related changes inside Web Application

Trong controller, sử dụng `@Valid` hoặc `@Validated` để áp dụng kiểm tra hợp lệ trên các đối tượng được truyền vào.

```
@RequestMapping(value = "/saveMsg", method = POST)
public String saveMessage(@Valid @ModelAttribute("contact") Contact contact, Errors errors){

    if(errors.hasErrors()){
        log.error("Contact form validation failed due to : " + errors.toString());
        return "contact.html";
    }
    contactService.saveMessageDetails(contact);
    return "redirect:/contact";
}
```

Ở ví dụ trên, sử dụng `@Valid` để áp dụng kiểm tra hợp lệ trên đối tượng `Contact` từ yêu cầu POST, xử lý lỗi (nếu có), lưu thông tin liên hệ và chuyển hướng người dùng đến một trang hiển thị thành công hoặc trang.



Section 8: Beans Web scopes inside Spring framework

1. Introduction to Spring Web Scopes

Trong Spring Framework, các Bean Web scopes được sử dụng để quản lý vòng đời của các Bean trong môi trường web. Các Web scopes xác định cách các Bean được tạo, duy trì và hủy bỏ theo các vòng đời của yêu cầu web. Dưới đây là một số Web scopes phổ biến trong Spring:

- **Request (@RequestScope)** - Mỗi yêu cầu web sẽ tạo ra một instance mới của Bean. Bean chỉ tồn tại trong phạm vi của yêu cầu hiện tại và sau đó được hủy bỏ..
- **Session (@SessionScope)** - Mỗi phiên làm việc (session) sẽ có một instance riêng của Bean. Bean tồn tại trong suốt phiên làm việc và được hủy bỏ khi phiên làm việc kết thúc.
- **Application (@ApplicationScope)** - Một instance của Bean được tạo ra cho toàn bộ ứng dụng web. Bean tồn tại trong suốt vòng đời của ứng dụng và chỉ được hủy bỏ khi ứng dụng web dừng hoặc khởi động lại.

2. Use Cases of Spring Web Scopes

Request Scope

- Spring tạo rất nhiều instance của bean này trong bộ nhớ của ứng dụng cho mỗi yêu cầu HTTP. Vì vậy, những loại Bean này tồn tại trong thời gian ngắn.
- Vì Spring tạo rất nhiều instance, vui lòng đảm bảo tránh tốn thời gian logic trong khi tạo instance.
- Có thể được xem xét cho các tình huống dữ liệu cần reset sau khi new request hay page refresh, v.v.

Session Scope

- Bean của Session Scope tồn tại lâu hơn và chúng ít bị thu gom rác hơn.
- Tránh giữ quá nhiều thông tin bên trong dữ liệu session vì nó ảnh hưởng đến hiệu suất. Không bao giờ lưu trữ thông tin nhạy cảm là tốt.
- Có thể được xem xét cho các tình huống trong đó cùng một dữ liệu cần được truy cập trên nhiều trang như thông tin người dùng.

Application Scope

- Trong phạm vi ứng dụng, Spring tạo một bean instance cho mỗi web application runtime.
- Nó tương tự như singleton scope, với một điểm khác biệt chính. Singleton scoped bean là singleton trên ApplicationContext trong đó application scoped bean là singleton trên mỗi ServletContext.
- Có thể được xem xét cho các tình huống mà chúng tôi muốn lưu trữ các giá trị Drop Down, Reference table sẽ không thay đổi đối với tất cả người dùng.

Section 9: Exception Handling using @ControllerAdvice & @ExceptionHandler (example30)

Trong Spring Framework, @ControllerAdvice và @ExceptionHandler là hai annotation quan trọng được sử dụng để xử lý ngoại lệ và kiểm soát hành vi của các Controller trong ứng dụng web.

- @ControllerAdvice: Đây là một annotation được sử dụng để đánh dấu một lớp là một Global Controller Advice trong ứng dụng. Một lớp được đánh dấu bằng @ControllerAdvice có thể chứa các phương thức xử lý ngoại lệ và logic chung áp dụng cho tất cả các Controller trong ứng dụng.
- @ExceptionHandler: Đây là một annotation được sử dụng để xác định phương thức xử lý ngoại lệ trong một Controller hoặc một Controller Advice. Khi một ngoại lệ xảy ra trong quá trình xử lý yêu cầu, phương thức được Annotation bởi @ExceptionHandler sẽ được gọi để xử lý ngoại lệ đó.

Với việc kết hợp sử dụng @ControllerAdvice và @ExceptionHandler, có thể xử lý các ngoại lệ chung trong toàn bộ ứng dụng hoặc xử lý ngoại lệ cụ thể trong từng Controller.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        // Xử lý ngoại lệ và trả về một ResponseEntity
        return new ResponseEntity<>("Something went wrong",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

@Controller
public class MyController {

    @GetMapping("/example")
    public ResponseEntity<String> example() throws Exception {
        // Ném một ngoại lệ
        throw new Exception("Some error occurred");
    }
}
```

Trong ví dụ trên, @ControllerAdvice được sử dụng để đánh dấu lớp GlobalExceptionHandler là một Global Controller Advice. Phương thức handleException() trong lớp này được đánh dấu bởi @ExceptionHandler(Exception.class), có nghĩa là nó sẽ được gọi khi xảy ra bất kỳ ngoại lệ nào trong quá trình xử lý yêu cầu.

Trong MyController, khi yêu cầu GET "/example" được gửi, một ngoại lệ Exception sẽ được ném. GlobalExceptionHandler sẽ được gọi và phương thức handleException() sẽ được thực thi để xử lý ngoại lệ và trả về một ResponseEntity với mã trạng thái 500.

Qua đó, @ControllerAdvice và @ExceptionHandler giúp xử lý ngoại lệ một cách linh hoạt và kiểm soát hành vi của các Controller trong ứng dụng Spring.

Section 10: Introduction to Spring Data & Spring Data JPA framework

1. Problems with Spring JDBC & how ORM frameworks solve these problems

Giới thiệu về JDBC

- JDBC hoặc Java Database Connectivity là một đặc điểm kỹ thuật từ Core Java cung cấp một bản tóm tắt tiêu chuẩn cho các ứng dụng Java để giao tiếp với các cơ sở dữ liệu khác nhau.
- API JDBC cùng với trình điều khiển cơ sở dữ liệu có khả năng truy cập cơ sở dữ liệu
- JDBC là framework cơ sở hoặc tiêu chuẩn cho các framework như Hibernate, Spring Data JPA, MyBatis, v.v.

Steps in JDBC to access DB

Chúng ta cần làm theo các bước dưới đây để truy cập DB bằng JDBC,

1. Load Driver Class
2. Lấy kết nối DB
3. Lấy một câu lệnh bằng cách sử dụng đối tượng connection
4. Thực hiện truy vấn
5. Xử lý tập kết quả
6. Đóng kết nối

Problem with JDBC

- Developers buộc phải tuân theo tất cả các bước được đề cập để thực hiện bất kỳ loại hoạt động nào với DB dẫn đến nhiều mã trùng lặp ở nhiều nơi. v
- Developers cần xử lý các checked exceptions sẽ ném ra khỏi API.
- JDBC phụ thuộc vào cơ sở dữ liệu

Chương trình mẫu sử dụng JDBC để lấy dữ liệu từ CSDL.

```
public Optional<Contact> findById(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        connection = DriverManager.getConnection("url", "username", "pwd");
        statement = connection.prepareStatement(
            "select id, name, message from contact");
        statement.setInt(1, id);
        resultSet = statement.executeQuery();
        Contact contact = null;
        if(resultSet.next()) {
            contact = new Contact(
                resultSet.getInt("id"),
                resultSet.getString("name"),
                resultSet.getString("message"));
        }
        return Optional.of(contact);
    } catch (SQLException e) {
        // ??? What should be done here ???
    } finally {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {}
        }
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {}
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {}
        }
    }
    return null;
}
```

2. Introduction to Spring Data

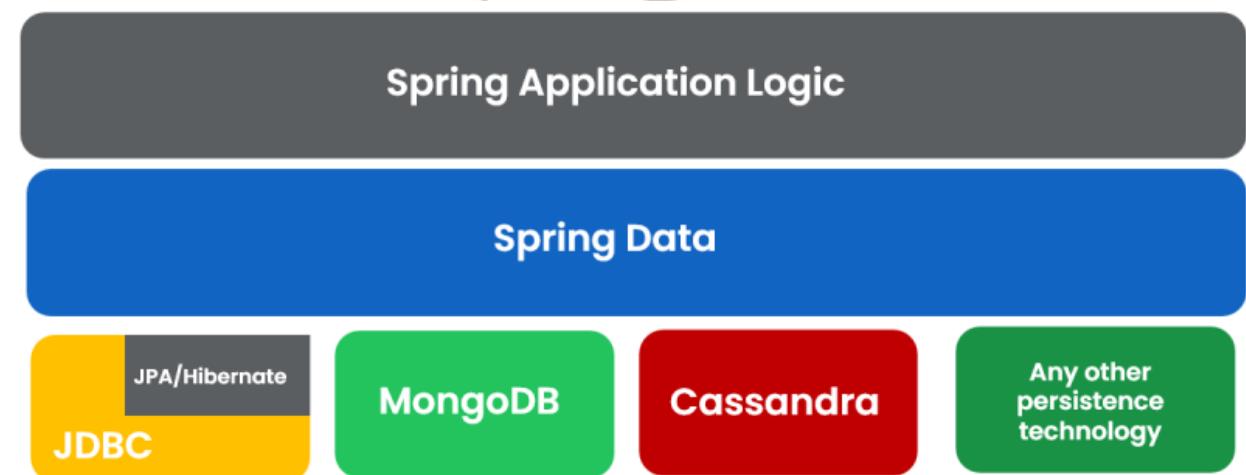
Spring Data là một dự án trong Spring Framework giúp đơn giản hóa việc làm việc với cơ sở dữ liệu trong ứng dụng Java. Nó cung cấp một tập hợp các công cụ, API và tích hợp với các công nghệ cơ sở dữ liệu khác nhau để thực hiện các tác vụ liên quan đến cơ sở dữ liệu một cách dễ dàng và hiệu quả.

Với Spring Data, bạn có thể làm việc với các cơ sở dữ liệu SQL như MySQL, PostgreSQL, Oracle, và cả cơ sở dữ liệu NoSQL như MongoDB, Redis. Nó cung cấp một cách tiếp cận chung để tương tác với các cơ sở dữ liệu này mà không cần phải viết mã SQL hoặc truy vấn cơ sở dữ liệu trực tiếp.

Spring Data tập trung vào việc cung cấp các tính năng chung như:

- **Repository Abstraction:** Spring Data cung cấp một cơ chế repository abstraction, cho phép bạn xác định các interface repository để thực hiện các tác vụ CRUD và truy vấn dữ liệu. Nó tự động cung cấp các phương thức thông qua khai báo tên phương thức và sử dụng các quy ước ánh xạ.
- **Query Methods:** Bằng cách sử dụng các phương thức trong interface repository, bạn có thể định nghĩa các truy vấn dữ liệu bằng cách sử dụng các quy tắc ánh xạ và cú pháp đặc biệt. Spring Data sẽ tự động tạo các truy vấn SQL tương ứng dựa trên các phương thức này.
- **Paging and Sorting:** Spring Data cung cấp hỗ trợ cho phân trang và sắp xếp kết quả truy vấn. Bạn có thể định nghĩa các thông số phân trang và sắp xếp trong các phương thức repository và Spring Data sẽ tự động áp dụng chúng vào truy vấn.
- **Native Queries:** Ngoài ra, Spring Data cho phép bạn định nghĩa các truy vấn SQL native trong repository interface nếu cần.

Spring Data cung cấp tích hợp với các công nghệ ORM như Hibernate, JPA (Java Persistence API) và cung cấp cơ chế tự động ánh xạ đối tượng vào cơ sở dữ liệu. Điều này giúp giảm bớt mã phức tạp và giúp bạn tập trung vào logic ứng dụng chính hơn.



3. Deepdive on Repository, CrudRepository, PagingAndSortingRepository, JpaRepository

Cho dù ứng dụng của bạn sử dụng công nghệ persistence nào, Spring Data cung cấp một bộ interface chung mà bạn mở rộng để xác định các persistence của ứng dụng. (Trong Java khi nói đến Persistence Data thường sẽ liên quan các chủ đề ánh xạ giữa class trong Java và table trong database, lưu trữ và truy vấn dữ liệu sử dụng SQL.)

Interface trung tâm trong Spring Data repository abstraction là Repository.

Repository:

- Interface cơ bản nhất trong Spring Data.
- Đây là một marker interface, không chứa bất kỳ phương thức cụ thể nào.
- Repository interface nên được định nghĩa bằng cách kế thừa từ interface này để đánh dấu rằng nó là một repository interface.

CrudRepository:

- CrudRepository là một interface con của Repository.
- Cung cấp các phương thức cơ bản để thực hiện các thao tác CRUD.
- Bao gồm các phương thức như save(), findById(), findAll(), delete() và nhiều phương thức khác để làm việc với dữ liệu.

ListCrudRepository

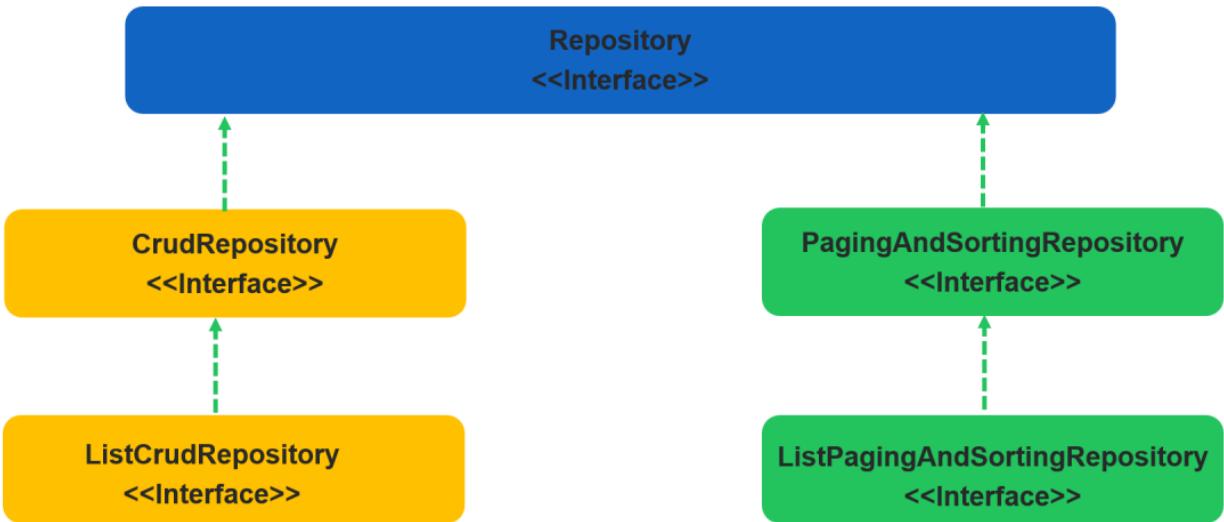
- Interface cho các hoạt động CRUD chung trên một repository cho một loại cụ thể, mở rộng CrudRepository và trả về List thay vì Iterable nếu có.

PagingAndSortingRepository:

- PagingAndSortingRepository là một interface con của CrudRepository.
- Bên cạnh các phương thức CRUD, nó cung cấp các phương thức để phân trang và sắp xếp kết quả truy vấn.
- Bao gồm các phương thức như findAll(Pageable), findAll(Sort) và các phương thức khác liên quan đến phân trang và sắp xếp.

JpaRepository:

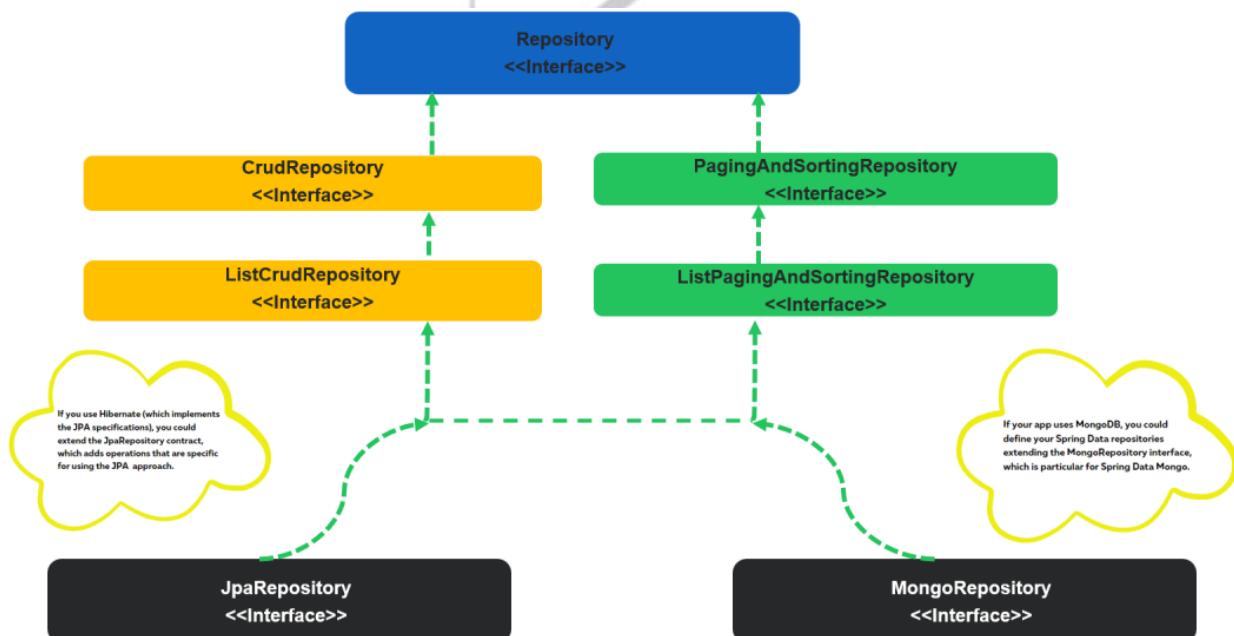
- JpaRepository là một interface con của PagingAndSortingRepository.
- Cung cấp các phương thức tiên tiến hơn để làm việc với dữ liệu, đặc biệt là khi sử dụng JPA (Java Persistence API).
- Bao gồm các phương thức như flush(), saveAndFlush(), deleteInBatch() và nhiều phương thức khác để thực hiện các thao tác cơ sở dữ liệu phức tạp hơn.



QUICK TIP

Bạn có biết,

- Chúng ta không nên nhầm lẫn giữa @Repository annotation và Spring Data Repository interface.
- Spring Data cung cấp nhiều interface và mở rộng lẫn nhau bằng cách tuân theo nguyên tắc được gọi là interface segregation - phân tách giao diện. Điều này giúp các ứng dụng mở rộng những gì chúng muốn.
- Một số mô-đun Spring Data có thể cung cấp các interface cụ thể cho công nghệ mà chúng đại diện. Ví dụ: sử dụng Spring Data JPA, bạn cũng có thể mở rộng trực tiếp giao diện JpaRepository và tương tự bằng cách sử dụng mô-đun Spring Data Mongo để ứng dụng của bạn cung cấp một interface cụ thể có tên MongoRepository.



4. Introduction to Spring Data JPA (example35)

Spring Data JPA là một phần mở rộng của Spring Framework, giúp đơn giản hóa việc làm việc với cơ sở dữ liệu quan hệ trong ứng dụng Java bằng cách sử dụng Java Persistence API (JPA). Nó cung cấp một tập hợp các công cụ và tích hợp với JPA để tạo ra các repository và thực hiện các thao tác CRUD (Create, Read, Update, Delete) và truy vấn dữ liệu.

Spring Data JPA tự động cung cấp các triển khai cho các interface repository mà bạn định nghĩa. Bằng cách sử dụng các quy ước ánh xạ và cú pháp đặc biệt, Spring Data JPA tự động tạo và thực thi các truy vấn SQL tương ứng dựa trên các phương thức trong repository interface.

Một số lợi ích của việc sử dụng Spring Data JPA bao gồm:

1. Giảm bớt mã phức tạp: Spring Data JPA giúp giảm bớt việc viết mã SQL và truy vấn cơ sở dữ liệu trực tiếp bằng cách sử dụng cú pháp đơn giản và khai báo các phương thức trong repository interface.
2. Tự động ánh xạ đối tượng: Spring Data JPA tự động ánh xạ các đối tượng Java vào cơ sở dữ liệu, giúp giảm bớt công việc phải viết mã ánh xạ đối tượng.
3. Hỗ trợ các chức năng JPA: Spring Data JPA hỗ trợ đầy đủ các tính năng của JPA như quan hệ đối tượng, quan hệ nhiều-nhiều, kết nối lười (lazy loading), v.v.
4. Tích hợp dễ dàng: Spring Data JPA dễ dàng tích hợp với các dự án Spring khác như Spring Boot, Spring MVC và Spring Security.

Dưới đây là maven dependency mà cần thêm vào bất kỳ dự án SpringBoot nào để sử dụng Spring Data JPA:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Chúng ta cần làm theo các bước dưới đây để truy vấn DB bằng cách sử dụng Spring Data JPA bên trong ứng dụng Spring Boot:

Bước 1: cần chỉ ra một lớp Java POJO là một lớp entity bằng cách sử dụng các annotation như @Entity, @Table, @Column

```
@Entity
@Table(name="contact_msg")
public class Contact extends BaseEntity{

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO, generator="native")
    @GenericGenerator(name = "native", strategy = "native")
    @Column(name = "contact_id")
    private int contactId;
```

Bước 2: cần tạo các interface cho một entity nhất định bằng cách mở rộng interface Repository được cung cấp bởi framework. Điều này giúp chúng tôi chạy các thao tác CRUD cơ bản trên bảng mà không cần triển khai các method

```
@Repository
public interface ContactRepository extends CrudRepository<Contact, Integer> {
}
```

Bước 3: Enable JPA và scanning bằng cách sử dụng Annotation @EnableJpaRepositories và @EntityScan

```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
public class EazyschoolApplication {
```

Bước 4: Trong các lớp service hoặc controller của bạn, tiêm vào dependency cho repository và sử dụng các phương thức đã được định nghĩa trong repository để truy vấn dữ liệu.

```
@Service
public class ContactService {

    @Autowired
    private ContactRepository contactRepository;

    public boolean saveMessageDetails(Contact contact){
        boolean isSaved = false;
        Contact savedContact = contactRepository.save(contact);
        if(null != savedContact && savedContact.getContactId() > 0) {
            isSaved = true;
        }
        return isSaved;
    }
}
```

5. Deep dive on derived query methods inside Spring Data JPA

Spring Data JPA cung cấp cho cách dễ dàng để tạo ra các truy vấn dựa trên tên phương thức của repository interface. Các truy vấn này được gọi là "Derived Query Methods". Điều này giúp cho việc tạo ra các truy vấn đơn giản và tiện lợi hơn cho các nhà phát triển.

Các phương thức truy vấn được xác định bởi tên phương thức. Tên phương thức cần tuân theo một số quy tắc:

- Tên phương thức bắt đầu với một động từ hành động như find, read, count, get,...
- Tiếp theo là By và sau là các thuộc tính của entity, ví dụ: findByFirstName, readByLastName,...
- Tên thuộc tính có thể được viết hoa để phân biệt chữ hoa chữ thường.

Các phương thức truy vấn cũng có thể được tùy chỉnh bằng cách sử dụng các từ khóa như And, Or, Between, Like, IsNotNull, In, NotIn, OrderBy, First, Top, Distinct, Limit và Offset.

Ví dụ, nếu có một entity là User với các thuộc tính id, firstName, lastName và age, có thể tạo ra các phương thức truy vấn sau:

```
// find persons by last name
List<Person> findByLastName(String lastName);
```

```
// find person by email
Person findByEmail(String email);
```

```
// find person by email and last name
Person findByEmailAndLastname(String email, String lastname);
```

Quick Tip

Bạn có biết một tên phương thức truy vấn dẫn xuất có hai thành phần chính được phân tách bằng từ khóa By đầu tiên.

1. Mệnh đề giới thiệu như find, read, query, count, hoặc get cho Spring Data JPA biết bạn muốn làm gì với phương thức. Mệnh đề này có thể chứa các biểu thức khác, chẳng hạn như Distinct để đặt một cờ riêng biệt cho truy vấn sẽ được tạo.
2. Mệnh đề tiêu chí(criteria clause) bắt đầu sau từ khóa By đầu tiên. By đầu tiên hoạt động như một dấu phân cách để chỉ ra điểm bắt đầu của tiêu chí truy vấn thực tế. Mệnh đề tiêu chí là nơi bạn xác định các điều kiện đối với các thuộc tính của entity và nối chúng với các từ khóa And và Or. Sử dụng readBy, getBy

và queryBy thay cho findBy sẽ hoạt động giống nhau. Ví dụ: readByEmail(String email) cũng giống như findByEmail(String email).

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
Distinct	findDistinctByLastnameAndFirstname	select distinct ... where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanOrEqual	findByAgeGreaterThanOrEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

6. Introduction of Auditing Support by Spring Data JPA (example 36)

Spring Data cung cấp hỗ trợ để theo dõi minh bạch ai đã tạo hoặc thay đổi một entity và thời điểm thay đổi xảy ra. Để hưởng lợi từ chức năng đó, bạn phải trang bị cho các lớp entity của mình auditing metadata có thể được xác định bằng cách sử dụng annotation hoặc bằng cách triển khai interface.

Ngoài ra, auditing phải được kích hoạt thông qua cấu hình Annotation hoặc cấu hình XML để đăng ký các thành phần cơ sở hạ tầng cần thiết.

Dưới đây là các bước cần phải được làm theo,

Bước 1: cần sử dụng annotation để chỉ ra các cột liên quan đến kiểm tra bên trong các table DB. Spring Data JPA cung cấp với một entity listener có thể được sử dụng để kích hoạt việc auditing thông tin. phải đăng ký AuditingEntityListener để được sử dụng cho tất cả các entity được yêu cầu.

```
@Data
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public class BaseEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdAt;
    @CreatedBy
    @Column(updatable = false)
    private String createdBy;
    @LastModifiedDate
    @Column(insertable = false)
    private LocalDateTime updatedAt;
    @LastModifiedBy
    @Column(insertable = false)
    private String updatedBy;
}
```



@CreatedDate, @CreatedBy, @LastModifiedDate, @LastModifiedBy are the key annotations that support JPA auditing

Bước 2: Thông tin liên quan đến ngày tháng sẽ được JPA tìm nạp từ máy chủ nhưng đối với Created By & UpdateBy, chúng tôi cần cho JPA biết cách tìm nạp thông tin đó thông tin bằng cách triển khai interface AuditorAware

```
@Component("auditAwareImpl")
public class AuditAwareImpl implements AuditorAware<String> {

    @Override
    public Optional<String> getCurrentAuditor() {
        return Optional.ofNullable(SecurityContextHolder.getContext()
            .getAuthentication().getName());
    }
}
```

Bước 3: Enable JPA auditing bằng annotation @EnableJpaAuditing

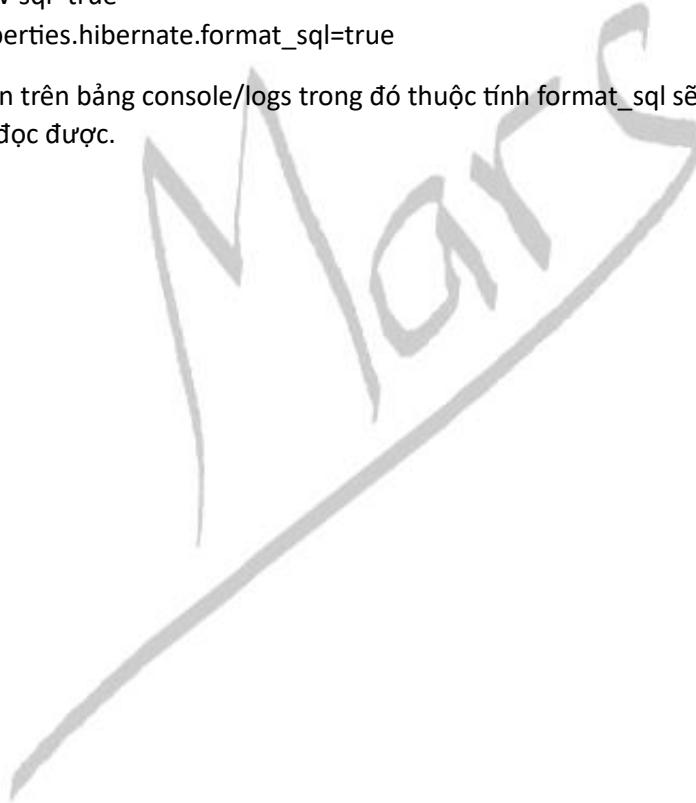
```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")
public class EazyschoolApplication {
```

Quick tip

Bạn có biết có thể in các truy vấn đang được Spring Data JPA tạo và thực thi bằng cách bật các thuộc tính bên dưới,

- spring.jpa.show-sql=true
- spring.jpa.properties.hibernate.format_sql=true

- show-sql sẽ in truy vấn trên bảng console/logs trong đó thuộc tính format_sql sẽ in các truy vấn theo kiểu thân thiện có thể đọc được.



Section 11: Building Custom Validations inside Spring MVC

Chúng ta đã thấy trước khi sử dụng các Bean validations như Max, Min, Size, v.v. có thể thực hiện validations trên dữ liệu nhận được. Bây giờ, hãy cố gắng custom validation cho các business requirement. Đối với điều tương tự, cần làm theo các bước dưới đây:

Step 1 : Suppose if we have a requirement to not allow some weak passwords inside our user registration form, we first need to create a custom annotation like below. Here we need to provide the class name where the actual validation logic present.

```
@Documented
@Constraint(validatedBy = PasswordStrengthValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface PasswordValidator {
    String message() default "Please choose a strong password";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Step 2 : We need to create a class that implements ConstraintValidator interface and overriding the isValid() method like shown below.

```
public class PasswordStrengthValidator implements
    ConstraintValidator<PasswordValidator, String> {

    List<String> weakPasswords;

    @Override
    public void initialize(PasswordValidator passwordValidator) {
        weakPasswords = Arrays.asList("12345", "password", "qwerty");
    }

    @Override
    public boolean isValid(String passwordField,
        ConstraintValidatorContext ctx) {
        return passwordField != null && (!weakPasswords.contains(passwordField));
    }
}
```

Step 3 : Finally we can mention the annotation that we created on top of the field inside a POJO class.

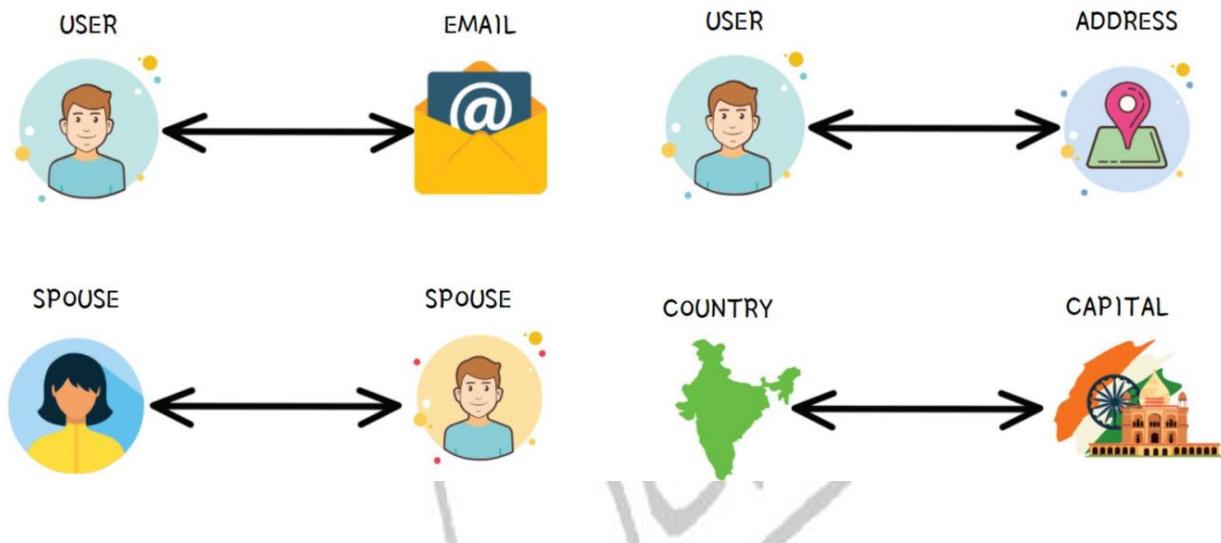
```
@NotBlank(message="Password must not be blank")
@Size(min=5, message="Password must be at least 5 characters long")
@PasswordValidator
private String pwd;
```

Section 12: Deep dive on OneToOne Relationship, Fetch Types, Cascade Types in ORM frameworks (example37)

1. Introduction to One to One Relationship inside ORM frameworks

Đầu tiên, mối quan hệ one-to-one là gì? Đó là mối quan hệ trong đó một bản ghi trong một entity (table) được liên kết với chính xác một bản ghi trong một entity (table) khác.

+ Dưới đây là một số ví dụ thực tế về mối quan hệ one-to-one:



2. Making One to One Relationship configurations inside entity classes

- Spring Data JPA cho phép các nhà phát triển xây dựng mối quan hệ one-to-one giữa các entity (entity) với các cấu hình đơn giản. Ví dụ: nếu chúng tôi muốn xây dựng mối quan hệ one-to-one giữa các entity Person and Address, thì chúng tôi có thể định cấu hình như được đề cập bên dưới.

```
@Data
@Entity
public class Person {

    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, targetEntity = Address.class)
    @JoinColumn(name = "address_id", referencedColumnName = "addressId", nullable = true)
    private Address address;
}
```

Trong Spring Data JPA, mối quan hệ one-to-one giữa hai entity được khai báo bằng cách sử dụng Annotation `@OneToOne`. Sử dụng nó, có thể cấu hình `FetchType`, `cascade`, `targetEntity`.

Annotation `@JoinColumn` được sử dụng để chỉ định chi tiết mối quan hệ foreign key column giữa 2 entity. "name" xác định tên của foreign key column, cho biết tên trường bên trong target entity class, `nullable` xác định foreign key column có thể là null hay không.

3. Making One to One Relationship configurations inside entity classes

JPA FETCH

Dựa trên các cấu hình **FETCH**(tìm nạp) mà nhà phát triển đã thực hiện, JPA cho phép các entity tải các đối tượng mà chúng có mối quan hệ.

Chúng ta có thể khai báo giá trị **fetch** trong các annotation `@OneToOne`, `@OneToMany`, `@ManyToOne` và `@ManyToMany`.

Các annotation này có một thuộc tính(attribute) được gọi là **fetch** để chỉ ra loại **FETCH** mà chúng tôi muốn thực hiện. Nó có hai giá trị hợp lệ: `FetchType.EAGER` và `FetchType.LAZY`

Với cấu hình **LAZY**, chúng tôi đang nói với JPA rằng chúng tôi muốn tải một cách lười biếng các entity quan hệ, vì vậy khi truy xuất một entity, các quan hệ của nó sẽ không được tải cho đến khi chúng tôi thử tham chiếu entity liên quan bằng phương thức getter. Ngược lại, với **EAGER**, nó cũng sẽ tải các entity quan hệ của nó.

Theo mặc định, tất cả các mối quan hệ `ToMany` đều là **LAZY**, trong khi các mối quan hệ `ToOne` là **EAGER**.

4. Deep dive on Fetch Types and Cascade Types in ORM frameworks

Cascade Types là một tính năng trong các framework ORM (Object-Relational Mapping) như Hibernate hay Spring Data JPA, cho phép tự động áp dụng các thao tác (cascading operations) trên các đối tượng liên quan khi thực hiện một thao tác trên đối tượng gốc. Điều này tiện lợi trong việc quản lý các mối quan hệ giữa các đối tượng và đảm bảo tính nhất quán và toàn vẹn dữ liệu.

Key points of Cascade Types

Intro to Cascade

- JPA cho phép truyền các thay đổi trạng thái của entity từ các thực thể entity Parent sang entity Child. Khái niệm này đang gọi Cascading trong JPA.
- Tùy chọn cấu hình cascade chấp nhận một mảng `CascadeTypes`.

Cascade Types

- `CascadeType.PERSIST`
- `CascadeType.MERGE`
- `CascadeType.REFRESH`
- `CascadeType.REMOVE`
- `CascadeTypeDETACH`
- `CascadeType.ALL`

Best Practices

- Cascading chỉ có ý nghĩa đối với các liên kết Parent - Child (trong đó quá trình chuyển đổi trạng thái của thực thể Parent được cascade (xếp tầng) cho các thực thể Child của nó). Cascading từ Child đến Parent không hữu ích và không được khuyến khích.
- Không có loại default cascade trong JPA. Theo mặc định, không có thao tác cascade.

Dưới đây là một số Cascade Types phổ biến trong ORM frameworks:

1. CascadeType.ALL: Áp dụng tất cả các thao tác (insert, update, delete) trên đối tượng gốc và các đối tượng liên quan.
2. CascadeType.PERSIST: Áp dụng thao tác insert trên đối tượng gốc và các đối tượng liên quan.
3. CascadeType.MERGE: Áp dụng thao tác update trên đối tượng gốc và các đối tượng liên quan.
4. CascadeType.REMOVE: Áp dụng thao tác delete trên đối tượng gốc và các đối tượng liên quan.
5. CascadeType.REFRESH: Áp dụng thao tác refresh (tải lại) trên đối tượng gốc và các đối tượng liên quan.
6. CascadeType.DETACH: Áp dụng thao tác detach (tách ra) trên đối tượng gốc và các đối tượng liên quan.
7. CascadeType.ALL_EXCEPT_DELETE: Áp dụng tất cả các thao tác trừ delete trên đối tượng gốc và các đối tượng liên quan.
8. CascadeType.REMOVE_ORPHAN: Áp dụng thao tác delete trên các đối tượng liên quan bị cô lập (orphans).

Các Cascade Types cho phép quản lý mối quan hệ tự động, giảm thiểu công việc lập trình viên và đảm bảo tính nhất quán của dữ liệu trong cơ sở dữ liệu. Tuy nhiên, cần cẩn thận khi sử dụng Cascade Types, vì việc áp dụng các thao tác trên quá nhiều đối tượng có thể ảnh hưởng đến hiệu suất và toàn vẹn dữ liệu.

Section 13: Deep dive on OneToMany, ManyToOne Relationships in ORM frameworks (example41, example42)

1. Introduction to OneToMany & ManyToOne mappings

Mối quan hệ One-to-Many (một-nhiều) và Many-to-One (nhiều-một) là hai mối quan hệ phổ biến trong các framework ORM (Object-Relational Mapping) như Hibernate hay Spring Data JPA, để thiết lập quan hệ giữa các entity.

a) Mối quan hệ One-to-Many:

- Trong mối quan hệ One-to-Many, một entity ở phía một (one) có thể có nhiều entity ở phía nhiều (many) tương ứng với nó.
- Trong lớp entity ở phía một, sử dụng annotation phù hợp (ví dụ: @OneToMany trong JPA) để định nghĩa mối quan hệ và chỉ định lớp entity ở phía nhiều.
- Trong lớp entity ở phía nhiều, sử dụng annotation phù hợp (ví dụ: @ManyToOne trong JPA) để định nghĩa mối quan hệ và chỉ định lớp entity ở phía một.
- Sử dụng thuộc tính phù hợp (ví dụ: mappedBy trong JPA) trong lớp entity ở phía một để chỉ định trường hoặc thuộc tính trong lớp entity ở phía nhiều mà liên kết đến nó.

b) Mối quan hệ Many-to-One:

- Trong mối quan hệ Many-to-One, nhiều entity ở phía nhiều có thể liên kết với một entity ở phía một.
- Trong lớp entity ở phía nhiều, sử dụng annotation phù hợp (ví dụ: @ManyToOne trong JPA) để định nghĩa mối quan hệ và chỉ định lớp entity ở phía một.
- Trong lớp entity ở phía một, sử dụng annotation phù hợp (ví dụ: @OneToMany trong JPA) để định nghĩa mối quan hệ và chỉ định lớp entity ở phía nhiều.

- Sử dụng thuộc tính phù hợp (ví dụ: mappedBy trong JPA) trong lớp entity ở phía nhiều để chỉ định trường hoặc thuộc tính trong lớp entity ở phía một mà liên kết đến nó.

Mỗi quan hệ One-to-Many và Many-to-One giúp quản lý các quan hệ phức tạp giữa các entity trong cơ sở dữ liệu và cung cấp khả năng truy xuất, thêm, sửa đổi và xóa dữ liệu một cách thuận tiện và nhất quán.



Spring Data JPA cho phép các nhà phát triển xây dựng mối quan hệ One-to-Many và Many-to-One giữa các entity với các cấu hình đơn giản. Dưới đây là các cấu hình mẫu giữa Class và Persons:

@ManyToOne

```
@Entity
public class Person extends BaseEntity{

    @ManyToOne(fetch = FetchType.LAZY, optional = true)
    @JoinColumn(name = "class_id", referencedColumnName = "classId", nullable = true)
    private EazyClass eazyClass;
}
```

- Annotation @ManyToOne được sử dụng để xác định một quan hệ Many-to-One giữa hai entity.
- Annotation @JoinColumn được sử dụng để chỉ định foreign key column.
- Chỉ cần cấu hình trên lớp Person

@OneToMany

```
@Entity
@Table(name = "class")
public class EazyClass extends BaseEntity{

    @OneToMany(mappedBy = "eazyClass", fetch = FetchType.LAZY,
               cascade = CascadeType.PERSIST, targetEntity = Person.class)
    private Set<Person> persons;
}
```

- Mỗi quan hệ One-to-Many giữa hai entity được xác định bằng cách sử dụng annotation **@OneToMany**.
- Nó cũng khai báo phần tử **mappedBy** để chỉ ra entity sở hữu mối quan hệ hai chiều.
- Thông thường, entity con là entity sở hữu mối quan hệ và entity mẹ chứa annotation **@OneToMany**.

Ví dụ:

Trước tiên, ta có lớp Author và lớp Book với mối quan hệ One-to-Many, trong đó một tác giả có thể viết nhiều sách:

```

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;

    // Constructors, getters, setters
}

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Constructors, getters, setters
}

```

Tuy nhiên, không phải lúc nào cũng cần sử dụng cả hai annotation này cùng nhau. Trong một số trường hợp, bạn có thể sử dụng chỉ một trong hai annotation, tùy thuộc vào loại quan hệ giữa các đối tượng trong mô hình dữ liệu của bạn. Ví dụ, nếu quan hệ giữa hai đối tượng là một-nhiều hai chiều, bạn có thể chỉ sử dụng `@OneToMany` trên cả hai đối tượng.

Quan trọng là hiểu rõ mục đích và tính chất của quan hệ giữa các đối tượng trong cơ sở dữ liệu của bạn và sử dụng các annotation `@OneToMany` và `@ManyToOne` một cách phù hợp để xác định và tạo quan hệ đó trong Java Spring JPA.

Section 14: Deep dive on ManyToMany Relationship & Configurations inside ORM frameworks

1. Introduction to ManyToMany

Mỗi quan hệ Many-to-Many (nhiều-nhiều) là một loại quan hệ giữa các đối tượng trong cơ sở dữ liệu, trong đó mỗi đối tượng trong một bảng có thể được liên kết với nhiều đối tượng trong bảng khác và ngược lại. Trong khung công cụ ORM (Object-Relational Mapping) như Java Spring JPA, bạn có thể sử dụng các annotation để thiết lập mối quan hệ Many-to-Many giữa các đối tượng.

Trong Java Spring JPA, để xác định mối quan hệ Many-to-Many, bạn sử dụng hai annotation `@ManyToMany` và `@JoinTable`.

Annotation `@ManyToMany` được đặt trên trường hoặc phương thức của cả hai đối tượng trong quan hệ. Nó chỉ định một quan hệ Many-to-Many giữa hai đối tượng.

Annotation `@JoinTable` được sử dụng để chỉ định bảng liên kết để lưu trữ thông tin quan hệ Many-to-Many. Nó xác định tên bảng và các foreign key column để liên kết hai bảng.



2. Implement ManyToMany configurations inside Entity classes

Spring Data JPA cho phép các nhà phát triển xây dựng mối quan hệ Many-To-Many giữa các entity với các cấu hình đơn giản.

```
@Entity
public class Person extends BaseEntity{

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @JoinTable(name = "person_courses",
        joinColumns = {
            @JoinColumn(name = "person_id", referencedColumnName = "personId")},
        inverseJoinColumns = {
            @JoinColumn(name = "course_id", referencedColumnName = "courseId")})
    private Set<Courses> courses = new HashSet<>();
}
```

Theo hai chiều của @ManyToMany, chỉ một entity có thể sở hữu mối quan hệ. Ở đây chọn Courses làm entity sở hữu, thường đề cập đến tham số mappedBy trên entity sở hữu.

```
@Entity
public class Courses extends BaseEntity{

    @ManyToMany(mappedBy = "courses", fetch = FetchType.EAGER
        , cascade = CascadeType.PERSIST)
    private Set<Person> persons = new HashSet<>();
}
```

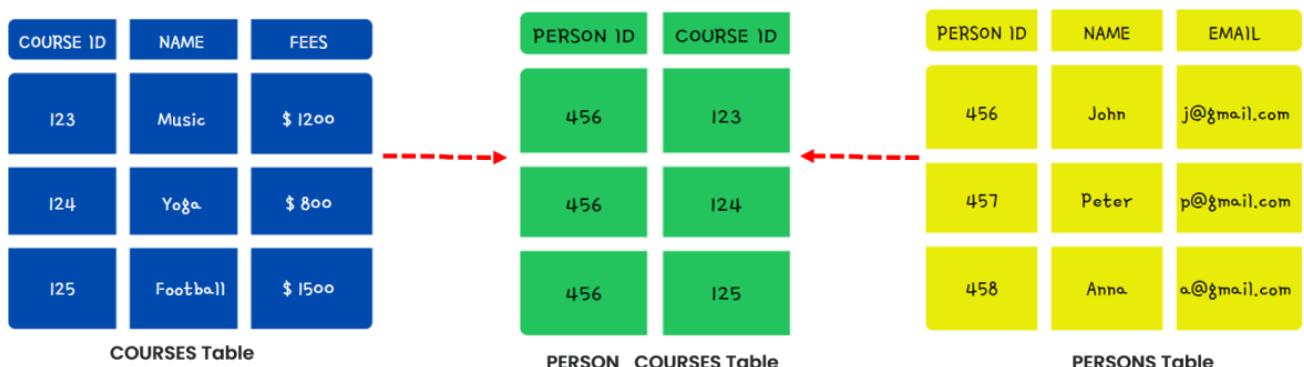
3. Why do we really need a third table in Many to Many relationship?

Trong mối quan hệ Many-to-Many, một bảng thứ ba, thường được gọi là "bảng liên kết" hoặc "bảng kết hợp," được sử dụng để đại diện cho mối quan hệ giữa hai entity. Bảng thứ ba này là cần thiết vì không thể thiết lập một sự tương ứng trực tiếp giữa hai entity trong cơ sở dữ liệu quan hệ.

Dưới đây là một số lý do vì sao cần có bảng thứ ba trong mối quan hệ Many-to-Many:

1. **Tính chất quan hệ:** Mỗi quan hệ Many-to-Many cho phép một entity có thể được liên kết với nhiều entity khác và ngược lại. Nếu không có bảng liên kết, sẽ khó khăn để đại diện cho mối quan hệ này trực tiếp trong cơ sở dữ liệu quan hệ vì mỗi entity thường có một khóa chính phải là duy nhất. Bảng liên kết cho phép lưu trữ các khóa ngoại của cả hai entity để ánh xạ nhiều quan hệ giữa chúng.
2. **Dữ liệu bổ sung:** Thường, có dữ liệu hoặc thuộc tính bổ sung được liên kết với mối quan hệ chính. Ví dụ, trong mối quan hệ Sách-Tác giả, bảng liên kết có thể bao gồm các thuộc tính như "ngày hợp tác" hoặc "vai trò của tác giả". Bảng liên kết cho phép lưu trữ các dữ liệu bổ sung này cụ thể cho mỗi quan hệ.
3. **Tính linh hoạt:** Việc sử dụng bảng liên kết mang lại tính linh hoạt cho mối quan hệ Many-to-Many. Nó cho phép thay đổi quan hệ mà không ảnh hưởng đến cấu trúc hoặc mô hình của các entity liên quan. Bạn có thể thêm hoặc xóa quan hệ giữa các entity chỉ bằng cách thao tác trên các bản ghi trong bảng liên kết.
4. **Hiệu suất và tối ưu hóa:** Bảng liên kết có thể cải thiện hiệu suất và tối ưu hóa truy vấn. Nó cung cấp một cấu trúc rõ ràng để hệ quản trị cơ sở dữ liệu làm việc và tối ưu hóa các truy vấn liên quan đến mối quan hệ Many-to-Many. Nó cho phép sắp xếp, lọc và kết hợp hiệu quả, cải thiện hiệu suất tổng thể của hệ thống.

Tóm lại, bảng thứ ba trong mối quan hệ Many-to-Many đóng vai trò là một bước trung gian giúp đại diện cho mối quan hệ và dữ liệu liên quan của nó. Nó cung cấp một giải pháp sạch và mở rộng cho việc mô hình hóa và truy vấn các mối quan hệ Many-to-Many trong cơ sở dữ liệu quan hệ.



Section 15: Sorting & Pagination inside Spring Data JPA

1. Sorting inside Spring Data JPA

Spring Data JPA cung cấp các cơ chế để thực hiện việc sắp xếp (sorting) dữ liệu trong các truy vấn cơ sở dữ liệu. Điều này giúp bạn dễ dàng thực hiện việc sắp xếp theo các thuộc tính cụ thể của đối tượng và theo thứ tự tăng dần hoặc giảm dần.

Để thực hiện việc sắp xếp bên trong Spring Data JPA, bạn có thể sử dụng một trong hai cách sau:

- Sử dụng từ khóa "OrderBy" trong tên phương thức: Bạn có thể thêm từ khóa "OrderBy" và sau đó liên kết các thuộc tính của đối tượng mà bạn muốn sắp xếp. Ví dụ:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByLastNameOrderByFirstNameAsc(String lastName);  
}
```

Trong ví dụ trên, phương thức `findByLastNameOrderByFirstNameAsc` sẽ trả về một danh sách các người dùng được sắp xếp theo trường `firstName` tăng dần khi `lastName` trùng khớp.

- Sử dụng `@Query` và câu truy vấn JPQL: Bạn có thể sử dụng annotation `@Query` và viết câu truy vấn JPQL(**JPA Query Language**) để chỉ định việc sắp xếp dữ liệu. Ví dụ:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("SELECT u FROM User u WHERE u.lastName = ?1 ORDER BY u.firstName  
ASC")  
    List<User> findByLastName(String lastName);  
}
```

Trong ví dụ này, sử dụng câu truy vấn JPQL để lấy danh sách người dùng có `lastName` trùng khớp và sắp xếp theo trường `firstName` tăng dần.

Ngoài ra, bạn có thể sử dụng từ khóa `Asc` hoặc `Desc` để chỉ định thứ tự sắp xếp tăng dần hoặc giảm dần. Ví dụ: `findByLastNameOrderByFirstNameDesc`.

Spring Data JPA cũng hỗ trợ việc sắp xếp theo nhiều thuộc tính cùng một lúc bằng cách liệt kê chúng. Ví dụ: `findByLastNameOrderByFirstNameAscLastNameDesc`.

Điều quan trọng là Spring Data JPA tự động xử lý việc sắp xếp dữ liệu dựa trên các thuộc tính mà bạn chỉ định, giúp giảm bớt công việc lặp lại và tăng tính linh hoạt trong việc thực hiện việc sắp xếp trong ứng dụng của bạn.

2. Static Sorting Và Dynamic Sorting

Trong Spring Data JPA, bạn có thể thực hiện sắp xếp (sorting) dữ liệu theo hai cách khác nhau: static sorting và dynamic sorting.

Static Sorting:

- Static sorting là phương pháp sắp xếp dữ liệu sử dụng một tiêu chí sắp xếp cố định và không thay đổi.
- Trong Spring Data JPA, bạn có thể sử dụng từ khóa "OrderBy" trong tên phương thức hoặc thông qua các câu truy vấn JPQL để thực hiện static sorting.
- Ví dụ: `findByLastNameOrderByFirstNameAsc` sẽ trả về danh sách người dùng được sắp xếp theo trường `firstName` tăng dần khi `lastName` trùng khớp.

Dynamic Sorting:

- Dynamic sorting là phương pháp sắp xếp dữ liệu dựa trên các tiêu chí sắp xếp được xác định tại thời điểm chạy.
- Trong Spring Data JPA, bạn có thể sử dụng đối tượng `Sort` để xác định các tiêu chí sắp xếp.
- Ví dụ: `Sort.by(Sort.Direction.ASC, "firstName")` sẽ sắp xếp theo trường `firstName` tăng dần.

Dynamic sorting cung cấp tính linh hoạt hơn vì bạn có thể xác định tiêu chí sắp xếp dựa trên nguồn dữ liệu đầu vào, như thứ tự được chọn bởi người dùng trong giao diện người dùng. Điều này cho phép ứng dụng của bạn thích ứng với các yêu cầu sắp xếp thay đổi mà không cần định nghĩa riêng từng phương thức sắp xếp cố định.

Để sử dụng dynamic sorting trong Spring Data JPA, bạn có thể sử dụng các phương thức như `findAll(Sort)` hoặc `findAll(Pageable)` trong repository. Ngoài ra, bạn có thể sử dụng các phương thức tạo truy vấn (QueryDSL, Specification, Criteria API) để xây dựng truy vấn có chứa dynamic sorting.

Ví dụ sử dụng dynamic sorting trong repository:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findAll(Sort sort);  
    Page<User> findAll(Pageable pageable);  
}
```

Ví dụ sử dụng dynamic sorting trong truy vấn tùy chỉnh:

```
public List<User> findByLastName(String lastName, Sort sort) {  
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();  
    CriteriaQuery<User> query = cb.createQuery(User.class);  
    Root<User> root = query.from(User.class);  
    query.where(cb.equal(root.get("lastName"), lastName));  
    query.orderBy(getOrderList(sort, cb, root)); // Hỗ trợ sắp xếp động  
    return entityManager.createQuery(query).getResultList();  
}  
  
private List<Order> getOrderList(Sort sort, CriteriaBuilder cb, Root<User> root) {  
    List<Order> orderList = new ArrayList<>();  
    if (sort != null) {  
        for (Sort.Order order : sort) {
```

```

        if (order.isAscending()) {
            orderList.add(cb.asc(root.get(order.getProperty())));
        } else {
            orderList.add(cb.desc(root.get(order.getProperty())));
        }
    }
    return orderList;
}

```

Static Sorting using derived query from method name

```
List<Person> findByNameOrderByAgeDesc(String name);
```

Static Sorting using @Query annotation with JPQL

```
@Query("SELECT p FROM person p WHERE p.age > ?1 ORDER BY p.name DESC")
List<Person> findByAgeGreaterThanJPQL(int age);
```

Static Sorting using @Query annotation with native query

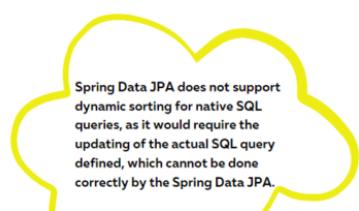
```
@Query(value = "SELECT * FROM person p WHERE p.name = :givenName ORDER BY p.age ASC",
       nativeQuery = true)
List<Person> findByGivenNameNativeSQL(@Param("givenName") String givenName);
```

Static Sorting using NamedQuery & NamedNativeQuery annotations

```
@NamedQuery(name = "Person.findByAgeGreaterThanNamedJPQL",
            query = "SELECT p FROM Person p WHERE p.age > :age ORDER BY p.name ASC")
@NamedNativeQuery(name = "Person.findAllNamedNativeSQL",
                  query = "SELECT * FROM person p ORDER BY p.age DESC")
@Entity
public class Person extends BaseEntity
```

Dynamic Sorting using Sort parameter. The Sort parameter can be passed with @Query, @NamedQuery annotations. Sort fields can be add dynamically as well based on the request params from the UI/API.

```
Sort sort = Sort.by("name").descending().and(Sort.by("age"));
List<Person> persons = personRepository.findByName("John", sort);
```



Spring Data JPA does not support dynamic sorting for native SQL queries, as it would require the updating of the actual SQL query defined, which cannot be done correctly by the Spring Data JPA.

3. Pagination inside Spring Data JPA

Spring Data JPA cung cấp các cơ chế để thực hiện phân trang (pagination) trong các truy vấn cơ sở dữ liệu. Phân trang là một kỹ thuật cho phép bạn lấy dữ liệu một cách chia nhỏ thành các trang, giúp hiển thị dữ liệu một cách hợp lý và tối ưu việc truy xuất dữ liệu.

Để thực hiện phân trang trong Spring Data JPA, bạn có thể sử dụng các phương thức của JpaRepository hoặc sử dụng đối tượng **Pageable**.

Phương thức của JpaRepository:

- Các phương thức `findAll(Pageable)` và `findAll(Sort)` trong JpaRepository cho phép bạn thực hiện phân trang dựa trên Pageable hoặc sắp xếp dựa trên Sort.
- Ví dụ: `Page<User> findAll(Pageable pageable)` sẽ trả về một trang dữ liệu người dùng dựa trên thông tin phân trang.

Đối tượng Pageable:

- Đối tượng Pageable được sử dụng để xác định các thông số liên quan đến phân trang, như số trang, số lượng bản ghi trên mỗi trang và thông tin sắp xếp.
- Bạn có thể tạo một đối tượng PageRequest để xác định các thông số phân trang:

```
Pageable pageable = PageRequest.of(pageNumber, pageSize,  
Sort.by("firstName").ascending());
```

Trong đó, `pageNumber` là số trang cần lấy, `pageSize` là số lượng bản ghi trên mỗi trang và `Sort` xác định tiêu chí sắp xếp (nếu cần).

Đối tượng Page:

- Đối tượng Page được trả về từ các phương thức phân trang và chứa thông tin về các bản ghi trên trang hiện tại, số lượng trang, tổng số bản ghi và các thông tin phân trang khác.
- Bạn có thể truy cập các thông tin của đối tượng Page để hiển thị dữ liệu phân trang:

```
List<User> users = page.getContent(); // Lấy danh sách bản ghi trên  
trang hiện tại  
int totalPages = page.getTotalPages(); // Tổng số trang  
long totalElements = page.getTotalElements(); // Tổng số bản ghi
```

Phân trang giúp bạn hiển thị dữ liệu một cách phân tán và tối ưu việc truy xuất dữ liệu từ cơ sở dữ liệu. Nó hữu ích khi làm việc với các bảng dữ liệu lớn hoặc khi cần hiển thị dữ liệu theo từng trang trong giao diện người dùng.

4. Pagination và kết hợp với Dynamic Sorting

Trong Spring Data JPA, bạn có thể kết hợp phân trang (pagination) với dynamic sorting để hiển thị dữ liệu phân trang và cho phép người dùng sắp xếp dữ liệu theo yêu cầu của mình.

Để kết hợp phân trang và dynamic sorting trong Spring Data JPA, bạn cần sử dụng đối tượng Pageable và Sort cùng nhau. Dưới đây là một ví dụ cụ thể:

```
public interface UserRepository extends JpaRepository<User, Long> {
    Page<User> findByLastName(String lastName, Pageable pageable);
}
```

Trong ví dụ trên, phương thức `findByLastName` sẽ trả về một trang dữ liệu người dùng có `lastName` trùng khớp, sử dụng phân trang thông qua đối tượng `Pageable`.

Khi sử dụng phân trang và dynamic sorting, bạn có thể tạo một đối tượng `PageRequest` và truyền nó vào phương thức truy vấn để xác định số trang, số lượng bản ghi trên mỗi trang và tiêu chí sắp xếp.

Ví dụ:

```
int pageNumber = 0; // Số trang (0 là trang đầu tiên)
int pageSize = 10; // Số lượng bản ghi trên mỗi trang
Sort sort = Sort.by("firstName").ascending(); // Tiêu chí sắp xếp (tăng dần theo firstName)

Pageable pageable = PageRequest.of(pageNumber, pageSize, sort);
Page<User> userPage = userRepository.findByLastName("Smith", pageable);
```

Trong ví dụ này, tạo một đối tượng `PageRequest` với số trang là 0, số lượng bản ghi trên mỗi trang là 10 và tiêu chí sắp xếp là tăng dần theo trường `firstName`. Sau đó, sử dụng đối tượng `Pageable` này trong phương thức `findByLastName` để lấy một trang dữ liệu người dùng có `lastName` là "Smith".

Kết quả trả về là một đối tượng `Page` chứa các bản ghi trên trang hiện tại, thông tin phân trang và số lượng bản ghi tổng cộng. Bạn có thể truy cập các thông tin này để hiển thị dữ liệu phân trang và sắp xếp tùy theo yêu cầu của người dùng.

Tổ hợp phân trang và dynamic sorting giúp bạn hiển thị dữ liệu một cách linh hoạt, tối ưu việc truy xuất dữ liệu và cung cấp khả năng sắp xếp theo yêu cầu của người dùng.

Below is an example where we are telling to JPA to fetch the first page by considering the total page size as 5

```
Pageable pageable = PageRequest.of(0, 5);
Page<Contact> msgPage = contactRepository.findByStatus("Open", pageable);
```

Below is an example where we are applying both pagination & sorting dynamically based on the input received

```
public Page<Contact> findMsgsWithOpenStatus(int pageNum, String sortField,
                                              String sortDir) {
    int pageSize = 5;
    Pageable pageable = PageRequest.of(pageNum - 1, pageSize,
        sortDir.equals("asc") ? Sort.by(sortField).ascending()
        : Sort.by(sortField).descending());
    Page<Contact> msgPage = contactRepository.findByStatus("Open", pageable);
    return msgPage;
}
```

Dynamic sorting is not supported by named queries. So while using Pagination along with Named Queries, make sure that `Pageable` instance does not contain a `Sort` object.

Section 16: Writing Custom Queries inside Spring Data JPA

1. Introduction to custom queries using @Query, @NamedQuery, @NamedNativeQuery & JPQL

Đối với các kịch bản truy vấn phức tạp, Spring Data JPA cho phép các nhà phát triển viết các truy vấn của riêng họ với sự trợ giúp của các chú thích bên dưới:

- @Query
- @NamedQuery
- @NamedNativeQuery

@Query Annotation

- **Annotation** @Query xác định các truy vấn trực tiếp trên các phương thức của repository. Điều này cho phép bạn hoàn toàn linh hoạt để chạy bất kỳ truy vấn nào mà không cần tuân theo các quy ước đặt tên phương thức.
- Với sự trợ giúp của chú thích @Query, có thể viết các truy vấn dưới dạng truy vấn JPQL hoặc SQL gốc.
- Khi viết một truy vấn SQL gốc thì cần đề cập đến nativeQuery = true bên trong chú thích @Query

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.age > :age")
    List<User> findByAgeGreaterThan(@Param("age") int age);
}
```

@NamedQuery

- @NamedQuery là một annotation được sử dụng để đặt tên cho một truy vấn JPQL và xác định truy vấn này trước thời điểm chạy.
- @NamedQuery được đặt trên entity class và được tham chiếu bằng tên khi thực hiện truy vấn.

```
@Entity
@NamedQuery(name = "Customer.findByAge", query = "SELECT c FROM Customer c
WHERE c.age >= :age")
public class Customer {
    //...
}
```

Trong ví dụ này, @NamedQuery được đặt tên là "Customer.findByAge" và truy vấn được định nghĩa là "SELECT c FROM Customer c WHERE c.age >= :age". Giá trị :age được sử dụng để truyền tham số cho truy vấn.

Sau đó, có thể sử dụng tên của @NamedQuery trong repository của để thực hiện truy vấn đó:

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long>
{
    List<Customer> findByAge(int age);
}
```

@NamedNativeQuery

- @NamedNativeQuery tương tự như @NamedQuery, nhưng được sử dụng cho truy vấn SQL native thay vì JPQL.

Ví dụ:

```
@Entity
@NamedNativeQuery(name = "User.findNativeByLastName", query = "SELECT * FROM
users WHERE last_name = :lastName", resultClass = User.class)
public class User {
    // ...
}
```

@Query example with positional parameters

```
// Positional parameters
@Query("SELECT c FROM Contact c WHERE c.status = ?1 AND c.name = ?2 ORDER BY c.createdAt DESC")
List<Contact> findByGivenQueryOrderByCreatedDesc(String status, String name);
```

@Query example with named parameters

```
// named parameters
@Query("SELECT c FROM Contact c WHERE c.status = :status AND c.name = :name ORDER BY c.createdAt DESC")
List<Contact> findByGivenQueryOrderByCreatedDesc(@Param("status") String status,
                                                @Param("name") String name);
```

Using @Query, we can also run UPDATE, DELETE, INSERT as well

```
@Transactional
@Modifying
@Query("UPDATE Contact c SET c.status = ?1 WHERE c.contactId = ?2")
int updateStatusById(String status, int id);
```

@NamedQuery example declared on top of the entity class

```
@Entity
@NamedQuery(name="Contact.findOpenMsgs",query = "SELECT c FROM Contact c WHERE c.status = :status")
public class Contact extends BaseEntity{
```

@NamedNativeQuery example declared on top of the entity class

```
@Entity
@NamedNativeQuery(name = "Contact.findOpenMsgsNative",
                  query = "SELECT * FROM contact_msg c WHERE c.status = :status",resultClass = Contact.class)
public class Contact extends BaseEntity{
```

2. Multiple named queries and named native queries

Trong Spring Data JPA, bạn có thể sử dụng multiple named queries và native queries để tạo các truy vấn tùy chỉnh đến cơ sở dữ liệu. Dưới đây là cách sử dụng chúng:

Multiple Named Queries:

Để sử dụng multiple named queries, bạn có thể định nghĩa nhiều annotation @NamedQuery trên entity class với các tên truy vấn khác nhau.

Ví dụ:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "User.findByLastName", query = "SELECT u FROM User
u WHERE u.lastName = :lastName"),
    @NamedQuery(name = "User.findByAge", query = "SELECT u FROM User u
WHERE u.age = :age")
})
public class User {
    // ...
}
```

Multiple named native queries

```
@Entity
@NamedNativeQueries({
    @NamedNativeQuery(name = "User.findByLastName", query =
"findByLastName", resultClass = User.class),
    @NamedNativeQuery(name = "User.findByAge", query = "findByAge",
resultClass = User.class)
})
public class User {
    // ...
}
```

Bây giờ bạn có thể sử dụng các named native queries đã định nghĩa trong repository interface hoặc EntityManager.

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastName(String lastName);
    List<User> findByAge(int age);
}
```

3. JPQL (Java Persistence Query Language):

Java Persistence Query Language (JPQL) là ngôn ngữ truy vấn hướng đối tượng độc lập với nền tảng được định nghĩa như một phần của đặc tả Java Persistence API (JPA).

JPQL được sử dụng để thực hiện các truy vấn đối với các entity được lưu trữ trong cơ sở dữ liệu quan hệ. Nó lấy cảm hứng rất nhiều từ SQL và các truy vấn của nó giống với các truy vấn SQL về cú pháp, nhưng hoạt động dựa trên các đối tượng entity JPA hơn là trực tiếp với các bảng cơ sở dữ liệu.

Hạn chế duy nhất của việc sử dụng JPQL là nó hỗ trợ một tập hợp con của tiêu chuẩn SQL. Vì vậy, nó có thể không phải là một lựa chọn tuyệt vời cho các truy vấn phức tạp.

JPQL Example

Dưới đây là một ví dụ về JPQL. Bạn có thể thấy ví dụ đang sử dụng tên entity và các trường có bên trong nó thay vì sử dụng tên bảng và cột.

```
@Query("SELECT c FROM Contact c WHERE c.contactId = ?1 ORDER BY c.createdAt DESC")
List<Contact> findByldOrderByCreatedDesc(long id);
```

4. EntityManager

Trong trường hợp bạn cần tùy chỉnh hơn nữa, bạn có thể sử dụng EntityManager để tạo và thực hiện các truy vấn native tương tự như sau:

```
@Autowired
private EntityManager entityManager;

public List<User> findUsersByLastName(String lastName) {
    Query query = entityManager.createNativeQuery("SELECT * FROM users WHERE
last_name = :lastName", User.class);
    query.setParameter("lastName", lastName);
    return query.getResultList();
}

public List<User> findUsersByAge(int age) {
    Query query = entityManager.createNativeQuery("SELECT * FROM users WHERE
age = :age", User.class);
    query.setParameter("age", age);
    return query.getResultList();
}
```

Section 17: Consuming Rest Services using Spring framework

1. Introduction to Consuming Rest Services inside Web Applications

Ngoài việc xây dựng Rest service, thường có thể cần sử dụng Rest service do các nhà cung cấp bên thứ ba khác cung cấp. Vì vậy, biết cách sử dụng các Rest service cũng quan trọng không kém. (**Consuming : tiêu thụ, sử dụng**)

+ Dưới đây là các cách tiếp cận được sử dụng phổ biến nhất được cung cấp bởi Spring framework,

- **OpenFeign** - Một công cụ được cung cấp bởi dự án Spring Cloud. Sử dụng điều này rất giống với cách xây dựng Repository với Spring Data JPA. Theo cách tương tự, chỉ cần viết interface chứ không cần mã triển khai.
- **RestTemplate** - Một công cụ nổi tiếng mà các developer đã sử dụng kể từ Spring 3 để gọi các REST endpoints. RestTemplate ngày nay thường được sử dụng trong các ứng dụng Spring. Nhưng điều này không được chấp nhận vì lợi ích của WebClient.
- **WebClient** - được tạo như một phần của mô-đun Spring Web Reactive và sẽ thay thế RestTemplate cổ điển. Điều này được giới thiệu để hỗ trợ tất cả các chế độ gọi Sync và Async (non-blocking)

2. Consuming Rest Services using OpenFeign

Để sử dụng các REST service bằng OpenFeign (gọi đến các API của các service khác hoặc back-end khác), cần làm theo các bước dưới đây.

Bước 1: Sau khi thêm tất cả các dependencie cần thiết bên trong pom.xml. cần tạo một proxy interface với tất cả các chi tiết xung quanh API mà sẽ sử dụng. Bên trong interface, cần tạo tên method khớp với các chi tiết của API method(example45) mà sẽ sử dụng:

```
@FeignClient(name = "contact", url = "http://localhost:8080/api/contact",
    configuration = ProjectConfiguration.class)
public interface ContactProxy {

    @RequestMapping(method = RequestMethod.GET, value = "/getMessagesByStatus")
    @Headers(value = "Content-Type: application/json")
    public List<Contact> getMessagesByStatus(@RequestParam("status") String status);

}
```

Bước 2: Nếu cần gửi chi tiết authentication, tạo bean với chi tiết username và password.

```
@Bean
public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("admin@eazyschool.com", "admin");
}
```

Bước 3: nên sử dụng đối tượng proxy để thực hiện call Rest như hình bên dưới:

```

@Autowired
ContactProxy contactProxy;

@GetMapping("/getMessages")
public List<Contact> getMessages(@RequestParam("status") String status) {
    return contactProxy.getMessagesByStatus(status);
}

```

3. Consuming Rest Services using RestTemplate

Bước 1: Sau khi thêm tất cả các dependencie cần thiết bên trong porn.xml. Cần tạo Bean RestTemplate với chi tiết authentication

```

@Bean
public RestTemplate restTemplate() {
    RestTemplateBuilder restTemplateBuilder =
        new RestTemplateBuilder();
    return restTemplateBuilder.basicAuthentication
        ("admin@eazyschool.com", "admin").build();
}

```

Bước 2: Sử dụng RestTemplate để gửi các yêu cầu HTTP đến dịch vụ REST khác.

```

@PostMapping("/saveMsg")
public ResponseEntity<Response> saveMsg(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    HttpHeaders headers = new HttpHeaders();
    headers.add("invocationFrom", "RestTemplate");
    HttpEntity<Contact> httpEntity = new HttpEntity<>(contact, headers);
    ResponseEntity<Response> responseEntity = restTemplate.exchange(uri, HttpMethod.POST,
        httpEntity, Response.class);
    return responseEntity;
}

```

4. Consuming WebClient using WebClient

Bước 1: Sau khi thêm tất cả các dependecie cần thiết bên trong porn.xml. Cần tạo Bean WebClient với chi tiết authentication

```
@Bean
public WebClient webClient() {
    return WebClient.builder()
        .filter(ExchangeFilterFunctions.
            basicAuthentication("admin@eazyschool.com", "admin"))
        .build();
}
```

Bước 2: Sử dụng **WebClient** để gửi các yêu cầu HTTP đến dịch vụ REST khác

```
@Autowired
WebClient webClient;

@PostMapping("/saveMessage")
public Mono<Response> saveMessage(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    return webClient.post().uri(uri)
        .header("invocationFrom", "WebClient")
        .body(Mono.just(contact), Contact.class)
        .retrieve()
        .bodyToMono(Response.class);
}
```

Section 18: Deep dive on Spring Data Rest & HAL Explorer

1. Spring Data Rest(example47)

Spring Data REST là một phần mở rộng của Spring Framework cho phép bạn nhanh chóng tạo ra một RESTful API từ các repository Spring Data. Nó cung cấp một cách tiếp cận dễ dàng để xây dựng các dịch vụ web RESTful mà không cần viết quá nhiều mã.

Spring Data REST tự động sinh ra các endpoint REST cho các repository Spring Data của bạn, điều này cho phép bạn thao tác với các tài nguyên trong repository thông qua các phương thức HTTP như GET, POST, PUT và DELETE. Ngoài ra, nó cung cấp các tính năng như phân trang, sắp xếp, lọc và tìm kiếm.

Để bắt đầu sử dụng Spring Data REST trong dự án, chỉ cần thêm dependency sau vào tệp pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Dưới đây là một ví dụ đơn giản về việc sử dụng Spring Data REST để tạo một API RESTful từ repository Spring Data.

Đầu tiên, hãy xem xét một ví dụ đơn giản về một đối tượng Book trong ứng dụng của.

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String author;

    // constructors, getters, setters
}
```

Tiếp theo, tạo một repository Spring Data cho đối tượng Book:

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> { }
```

Sau đó, chỉ cần thêm annotation `@RepositoryRestResource` vào repository để kích hoạt Spring Data REST cho nó:

```
@RepositoryRestResource(collectionResourceRel = "books", path = "books")
public interface BookRepository extends JpaRepository<Book, Long> { }
```

Bây giờ, khi chạy ứng dụng, Spring Data REST sẽ tự động tạo các endpoint RESTful cho đối tượng Book. Ví dụ, có thể truy cập danh sách tất cả các sách thông qua đường dẫn /books và chi tiết một cuốn sách cụ thể thông qua /books/{id}.

Ngoài ra, Spring Data REST cung cấp các tính năng như phân trang, sắp xếp, lọc và tìm kiếm một cách tự động. Ví dụ, có thể sắp xếp danh sách sách theo tiêu đề bằng cách sử dụng GET /books?sort=title hoặc lọc các sách của một tác giả cụ thể bằng cách sử dụng GET /books?author=John.

Chúng ta cũng có thể thực hiện các thao tác CRUD thông qua các phương thức HTTP như GET, POST, PUT và DELETE trên các endpoint tương ứng.

2. HAL Explorer

HAL (Hypertext Application Language) Explorer là một công cụ giao diện người dùng được tích hợp sẵn trong Spring Data REST. Nó cung cấp một giao diện web đơn giản để khám phá và tương tác với các tài nguyên RESTful được tạo bởi Spring Data REST.

HAL Explorer cho phép bạn xem danh sách các tài nguyên có sẵn, tạo, chỉnh sửa và xóa tài nguyên, cũng như thực hiện các thao tác CRUD thông qua giao diện người dùng. Nó tự động tạo các biểu mẫu đơn giản để bạn điền thông tin và gửi các yêu cầu HTTP tương ứng.

Giao diện người dùng HAL Explorer được thiết kế để dễ sử dụng và trực quan, giúp bạn nhanh chóng tìm hiểu và tương tác với các API RESTful được tạo bởi Spring Data REST.

Để bắt đầu sử dụng HAL explorer cùng với Spring Data REST trong dự án, chỉ cần thêm phần phụ sau vào tệp pom.xml của mình:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-hal-explorer</artifactId>
</dependency>
```

Bạn có biết có thể thay đổi đường dẫn mặc định được hiển thị bởi Spring Data REST bằng cách sử dụng các cấu hình bên dưới.

Điều này sẽ thay đổi đường dẫn mặc định theo cấu hình của bạn.

```
spring.data.restbasePath=/data-api
```

spring.data.rest.basePath=/data-api

HAL Explorer Theme Layout About

Edit Headers /data-api/ Go!

Links

Relation	Name	Title	HTTP Request	Doc
	holidays			
	contacts			
	roles			
	easyClasses			
	courses			
	persons			
	addresses			
	profile			

Response Status
200 (OK)

Response Headers

cache-control	no-cache, no-store, max-age=0, must-revalidate
connection	keep-alive
content-type	application/hal+json
date	Fri, 21 Jan 2022 02:16:21 GMT
expires	0
keep-alive	timeout=60
pragma	no-cache
transfer-encoding	chunked
vary	Origin, Access-Control-Request-Method, Access-Control-Request-Headers
x-content-type-options	nosniff
x-frame-options	DENY
x-xss-protection	1; mode=block

Response Body

```
{  
  "links": {  
    "holidays": {  
      "href": "http://localhost:8080/data-api/holidays"  
    },  
    "contacts": {  
      "href": "http://localhost:8080/data-api/contacts?page_size=100"  
    },  
    "roles": {  
      "href": "http://localhost:8080/data-api/roles?page_size=100"  
    },  
    "easyClasses": {  
      "href": "http://localhost:8080/data-api/easyClasses?page_size=100"  
    },  
    "courses": {  
      "href": "http://localhost:8080/data-api/courses?page_size=100"  
    },  
    "persons": {  
      "href": "http://localhost:8080/data-api/persons?page_size=100"  
    },  
    "addresses": {  
      "href": "http://localhost:8080/data-api/addresses?page_size=100"  
    },  
    "profile": {  
      "href": "http://localhost:8080/data-api/profile"  
    }  
  }  
}
```

Securing Spring Data Rest APIs & HAL Explorer:

```
http.csrf().ignoringRequestMatchers("/saveMsg").ignoringRequestMatchers("/public/**")  
    .ignoringRequestMatchers("/api/**").ignoringRequestMatchers("/data-api/**").and()  
    .authorizeHttpRequests()  
    .requestMatchers("/data-api/**").authenticated()
```

Section 19: Logging Configurations inside SpringBoot

1. Introduction to Logging inside SpringBoot (example48)

Theo mặc định, không phải lo lắng về việc logging nếu chúng tôi đang sử dụng Spring Boot. Hầu hết các cấu hình và logging do chính Spring Boot thực hiện.

Thông thường có các loại sau:

FATAL (Logback doesn't have)

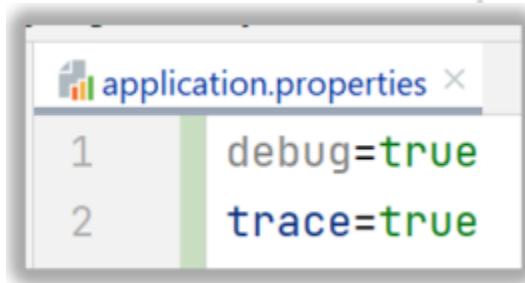
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Logging Types

- Trong Java, chúng tôi có nhiều logging framework như Java Util logging, Log4J2, SLF4J, Logback. Theo mặc định, nếu bạn sử dụng "Starters", Logback được sử dụng để ghi logging.
- Appropriate Logback thích hợp cũng được bao gồm trong Spring Boot để đảm bảo rằng các thư viện phụ thuộc sử dụng Java Util Logging, Commons Logging, Log4J hoặc SLF4J đều hoạt động chính xác.
- Theo mặc định, các thông báo ERROR-level, WARN-level, và INFO-level được ghi lại. Nhưng chúng tôi có thể thay đổi chúng dựa trên các yêu cầu và môi trường.

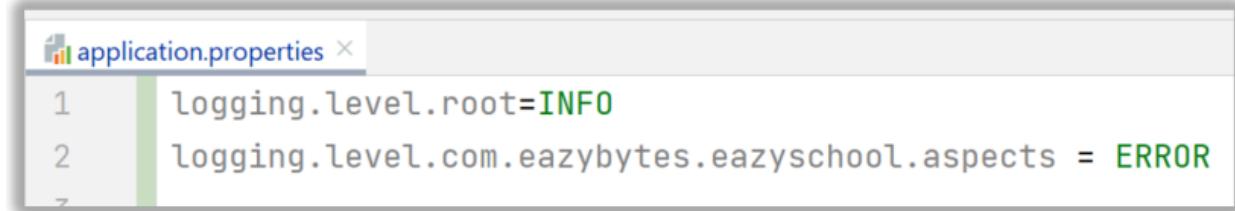
2. Logging configurations for SpringBoot framework code

Chúng ta có thể kích hoạt debug logging or trace logging bằng cách đề cập đến các thuộc tính bên trong tệp application.properties



```
application.properties
1 debug=true
2 trace=true
```

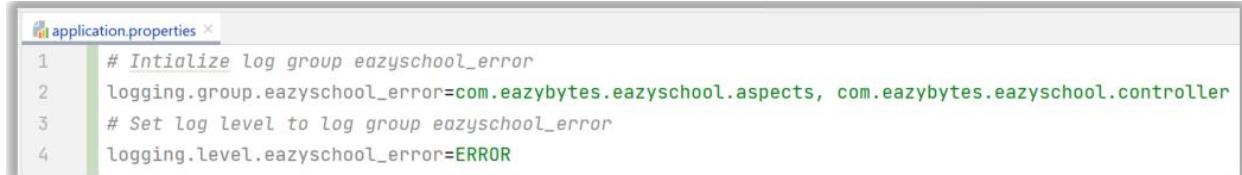
nếu cần, bạn có thể kiểm soát việc ghi logging ở cấp package bằng cách đề cập đến các thuộc tính bên trong tệp application.properties:



```
application.properties
1 logging.level.root=INFO
2 logging.level.com.eazybytes.eazyschool.aspects = ERROR
```

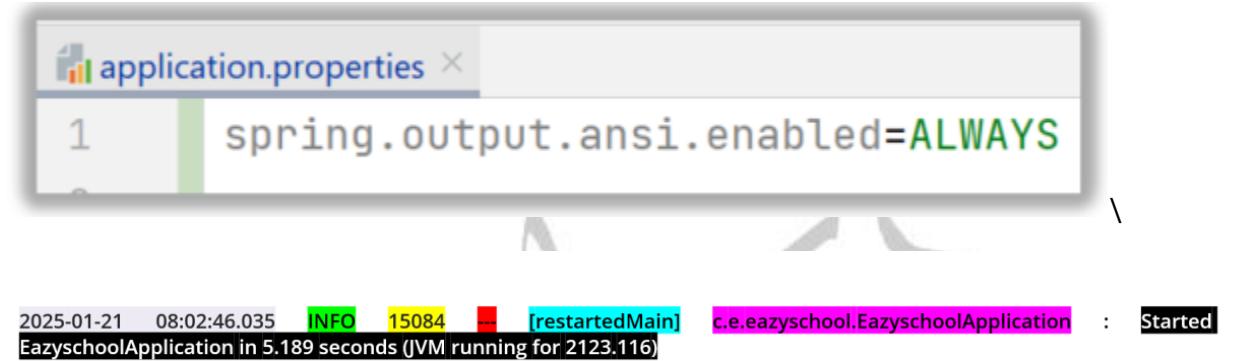
Việc có thể nhóm các log có liên quan lại với nhau để có thể cấu hình tất cả chúng cùng một lúc thường rất hữu ích. Ví dụ: có thể có yêu cầu thay đổi cấp độ logging cho tất cả các package liên quan đến dự án

của mình rất thường xuyên. Để giải quyết vấn đề này, Spring Boot cho phép xác định các logging group. Dưới đây là cấu hình mẫu:



```
# Initialize log group easyschool_error
logging.group.easyschool_error=com.eazybytes.easyschool.aspects, com.eazybytes.easyschool.controller
# Set log level to log group easyschool_error
logging.level.easyschool_error=ERROR
```

Nếu thiết bị hỗ trợ ANSI, color output sẽ được sử dụng để hỗ trợ khả năng đọc.



```
spring.output.ansi.enabled=ALWAYS
```

2025-01-21 08:02:46.035 INFO 15084 --- [restartedMain] c.e.easyschool.EasyschoolApplication : Started EasyschoolApplication in 5.189 seconds (JVM running for 2123.116)

- ✓ Date and Time: Millisecond precision and easily sortable.
- ✓ Log Level: ERROR, WARN, INFO, DEBUG, or TRACE.
- ✓ Process ID.
- ✓ A --- separator to distinguish the start of actual log messages.
- ✓ Thread name: Enclosed in square brackets (may be truncated for console output).
- ✓ Logger name: This is usually the source class name (often abbreviated).
- ✓ The log message.

Theo mặc định, Spring Boot chỉ ghi log vào console và không ghi log vào file. Nếu bạn muốn ghi các log file của console output, có thể tạo một file có tên logback.xml bên trong class path.

Quick Tip

Lombok có nhiều annotation khác nhau để giúp developers ghi logging dựa trên logging framework đang được sử dụng:

```
@Slf4j
public class LogExample {

}

public class LogExample {
    private static final org.slf4j.Logger log =
        org.slf4j.LoggerFactory.getLogger(LogExample.class);
}
```

Tương tự, sử dụng @CommonsLog, @Log4j2 sẽ tạo log variable từ lớp thư viện tương ứng.

Section 20: Properties Configuration & Profiles inside SpringBoot

1. Introduction to Externalized properties inside SpringBoot Web Applications

Spring Boot cho phép bạn cấu hình bên ngoài để bạn có thể làm việc với cùng mã ứng dụng trong các môi trường khác nhau. Bạn có thể sử dụng nhiều nguồn cấu hình bên ngoài, bao gồm Java properties files, YAML files, environment variables, và command-line arguments.

Theo mặc định, Spring Boot tìm kiếm các cấu hình hoặc thuộc tính bên trong application.properties/yaml có trong classpath. Nhưng cũng có thể có các tệp thuộc tính khác và SpringBoot đọc từ chúng.

Config/Properties Preferences

Spring Boot sử dụng một thứ tự rất cụ thể được thiết kế để phép ghi đè hợp lý các giá trị. Các Properties được xem xét theo thứ tự sau (với các giá trị từ các mục thấp hơn ghi đè lên các mục trước đó)

- Properties present inside files like application.properties
- OS Environmental variables
- Java System properties (System.getProperties())
- JNDI attributes from java:comp/env.
- ServletContext init parameters
- ServletConfig init parameters.
- Command line arguments.

2. Reading properties using @Value annotation

@Value annotation được sử dụng để chú thích các thuộc tính trong class hoặc các tham số đầu vào của method hay constructor để đặt tiêm các giá trị tương ứng được cấu hình trong các tệp cấu hình, ngoài ra nó còn được sử dụng để đặt giá trị mặc định cho các thuộc tính hoặc tham số đầu vào của method, constructor.

Default Value

Trường hợp sử dụng cơ bản nhất của @Value annotation là đặt giá trị mặc định cho các thuộc tính trong class.

```
@Value("John")
private String trainee;

@Value("100")
private int hoursOfCode;

@Value("true")
private boolean passedAssesmentTest;
```

Một điểm cần lưu ý là @Value chỉ nhận các giá trị kiểu String, sau đó những giá trị này sẽ được chuyển đổi sang kiểu dữ liệu tương ứng với kiểu dữ liệu của thuộc tính đã khai báo.

Spring Environment

Việc sử dụng @Value annotation để inject các giá trị được đặt trong các file cấu hình là một trong những cách sử dụng phổ biến nhất trong các ứng dụng Spring chạy trong thực tế.

Đầu tiên, đặt các giá trị trong file cấu hình chẵng hạn như application.properties hoặc application.yml tùy thuộc vào định dạng mà dự án đang sử dụng.

Giả sử có 3 thuộc tính được đặt giá trị trong file application.properties như thế này.

```
car.brand=Audi
car.color=Red
car.power=150
```

Để inject các giá trị này vào các thuộc tính trong một class có thể sử dụng cú pháp như sau:

```
@Value("${car.brand}")
private String brand;

@Value("${car.color}")
private String color;

@Value("${car.power}")
private int power;
```

Ngoài ra có thể đặt giá trị mặc định cho các thuộc tính này trong trường hợp một giá trị tương ứng được đặt trong tệp application.properties.

```
@Value("${car.type:Sedan}")
private String type;
```

System Variables

Cũng có thể truy cập các thuộc tính của hệ thống được Spring sử dụng khi khởi chạy ứng dụng:

```
@Value("${user.name}")
// Or
@Value("${username}")
private String userName;

@Value("${number.of.processors}")
// Or
@Value("${number_of_processors}")
private int numberOfProcessors;

@Value("${java.home}")
private String java;
```

@Value với biểu thức SpEL

Trong @Value có thể sử dụng biểu thức SpEL để lấy các giá trị.

Giả sử nếu có thuộc tính system được đặt tên là priority, thì giá trị của nó sẽ được áp dụng là:

```
@Value("#{systemProperties['priority']}")
private String spelValue;
```

Nếu không đặt giá trị cho thuộc tính systemProperties.priority thì giá trị NULL sẽ được đặt cho spelValue.

Nếu muốn đặt giá trị mặc định thay vì NULL có thể làm như sau:

```
@Value("#{systemProperties['priority'] ?: 'some default'}")
private String spelSomeDefault;
```

Hoặc nếu có một tập giá trị được ngăn cách bởi dấu phẩy thì có thể sử dụng biểu thức SpEL để tách chuỗi và trả về một danh sách các giá trị tương ứng.

```
@Value("#{${listOfValues}'.split(',')}")
private List<String> valuesList;
```

@Value với Constructor Injection

Có thể sử dụng @Value trên các tham số đầu vào của một constructor.

```
public class PriorityProvider {  
  
    private String priority;  
  
    @Autowired  
    public PriorityProvider(@Value("${priority:normal}") String priority) {  
        this.priority = priority;  
    }  
}
```

Bây giờ, giá trị priority sẽ tương ứng với priority:normal được chỉ định trong application.properties.

@Value với Setter injection

Cũng có thể sử dụng @Value trên các tham số đầu vào của một Setter method injection.

```
public class CollectionProvider {  
  
    private List<String> values = new ArrayList<>();  
  
    @Autowired  
    public void setValues(@Value("#{'${listOfValues}'.split(',')}")  
List<String> values) {  
    this.values.addAll(values);  
}  
  
    // standard getter  
}
```

Ở trên đã sử dụng biểu thức SpEL để đưa danh sách các giá trị vào phương thức setValues.

@Value annotation với Map

Có thể tiêm các thuộc tính có kiểu dữ liệu Map thông qua @Value annotation. Để làm được điều này đầu tiên cần định nghĩa một thuộc tính dạng key-value trong file application.properties.

```
valuesMap={key1: '1', key2: '2', key3: '3'}
```

Sau đó, tiêm valuesMap vào thuộc tính có kiểu dữ liệu là **Map** như sau:

```
@Value("#${valuesMap}")  
private Map<String, Integer> valuesMap;
```

Cũng có thể sử dụng một trong những giá trị của một key tương ứng bằng cách chỉ định rõ giá trị của key tương ứng trong @Value annotation.

```
@Value("#${valuesMap}.key1")  
private Integer valuesMapKey1;
```

Nếu không chắc chắn một key có tồn tại trong tệp cấu hình hay không thì có thể sử dụng biểu thức an toàn để nó giúp đặt giá trị thành NULL thay vì ném ra một exception.

Ngoài ra có thể đặt giá trị mặc định trong trường hợp không có các giá trị tương ứng của một Map hay một key tương ứng được định nghĩa trong file application.properties.

```
@Value("#{${unknownMap : {key1: '1', key2: '2'}}}")
private Map<String, Integer> unknownMap;
```

```
@Value("#{${valuesMap}['unknownKey'] ?: 5}")
private Integer unknownMapKeyWithDefaultValue;
```

Các phần tử của Map có thể lọc trước khi tiêm vào bằng cách sử dụng biểu thức SpEL.

Giả sử loại bỏ các phần tử có giá trị bé hơn 1.

```
@Value("#{${valuesMap}.?>'1'}}")
private Map<String, Integer> valuesMapFiltered;
```

Cũng có thể tiêm tất cả các thuộc tính hệ thống được Spring sử dụng vào một Map với cú pháp ngắn gọn như sau:

```
@Value("#{systemProperties}")
private Map<String, String> systemPropertiesMap;
```

3. Reading properties using Environment interface

Để đọc các thuộc tính sử dụng giao diện Environment trong Spring Framework, bạn có thể thực hiện các bước sau:

1. Đảm bảo rằng bạn đã cấu hình tệp tin cấu hình, ví dụ như application.properties hoặc application.yml, và chúng đã được đặt trong đường dẫn classpath.
2. Tiếp theo, bạn cần tiêm Environment vào lớp của mình bằng cách sử dụng chú thích @Autowired hoặc @Inject.

Ví dụ:

```
@Component
public class MyComponent {
    @Autowired
    private Environment environment;

    // ...
}
```

Bây giờ, bạn có thể sử dụng phương thức getProperty của Environment để đọc giá trị của thuộc tính.

```
String myProperty = environment.getProperty("my.property");
```

Trong ví dụ trên, giá trị của thuộc tính my.property trong tệp tin cấu hình sẽ được đọc và gán vào biến myProperty.

Bạn cũng có thể cung cấp giá trị mặc định cho thuộc tính nếu nó không tồn tại trong tệp tin cấu hình bằng cách sử dụng phương thức getProperty với tham số thứ hai là giá trị mặc định.

```
String myProperty = environment.getProperty("my.property", "default value");
```

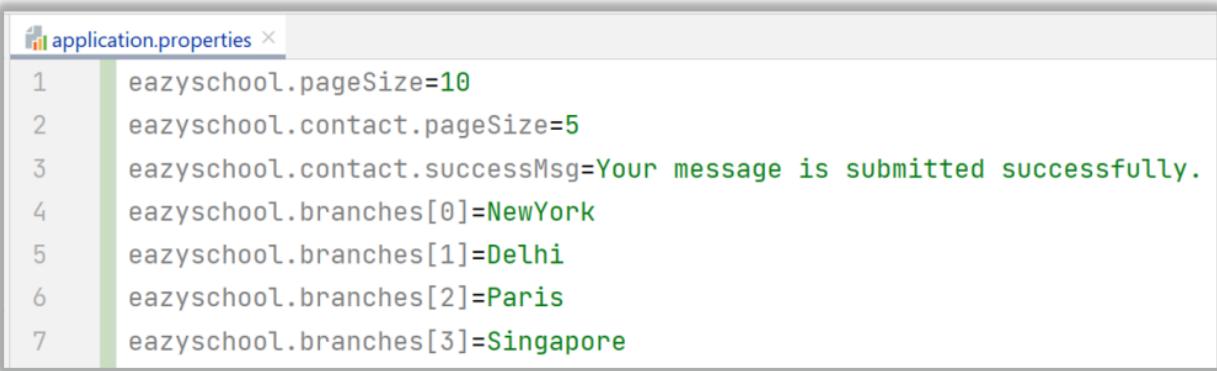
Nếu thuộc tính my.property không tồn tại trong tệp tin cấu hình, giá trị mặc định "default value" sẽ được sử dụng.

Điều này cho phép bạn đọc các thuộc tính từ tệp tin cấu hình bằng cách sử dụng giao diện Environment trong Spring Framework.

4. Reading properties using `@ConfigurationProperties`

SpringBoot cho phép tải tất cả các thuộc tính hợp lý với nhau vào một java bean. Đối với điều tương tự, có thể sử dụng chú thích `@ConfigurationProperties` trên đầu java bean bằng cách cung cấp giá trị tiền tố(prefix).

Bước 1: tạo các thuộc tính như bên dưới có tiền tố giống như 'eazyschool'



```
application.properties
1 eazyschool.pageSize=10
2 eazyschool.contact.pageSize=5
3 eazyschool.contact.successMsg=Your message is submitted successfully.
4 eazyschool.branches[0]=NewYork
5 eazyschool.branches[1]=Delhi
6 eazyschool.branches[2]=Paris
7 eazyschool.branches[3]=Singapore
```

Bước 2: Tạo một bean như bên dưới với tất cả các chi tiết cần thiết



```
@Component("eazySchoolProps")
@Data
@PropertySource("classpath:some.properties")
@ConfigurationProperties(prefix = "eazyschool")
@Validated
public class EazySchoolProps {

    @Min(value=5, message="must be between 5 and 25")
    @Max(value=25, message="must be between 5 and 25")
    private int pageSize;
    private Map<String, String> contact;
    private List<String> branches;
}
```

`@Validated` là một chú thích được sử dụng để kiểm tra và xác nhận tính hợp lệ của các đối số hoặc thuộc tính trong Spring Bean. Khi bạn áp dụng chú thích `@Validated` trên một lớp hoặc một phương thức, Spring sẽ kiểm tra các ràng buộc và quy tắc hợp lệ áp dụng cho các trường hợp sử dụng của đối số hoặc thuộc tính.

`@PropertySource` là một chú thích được sử dụng để chỉ định tệp tin cấu hình chứa các giá trị thuộc tính. Nó được sử dụng trong cấu hình của Spring để chỉ định tệp tin cấu hình mà Spring sẽ đọc để lấy giá trị thuộc tính.

Bước 3: Cuối cùng, chúng ta có thể thêm bean mà chúng ta đã tạo ở bước trước và bắt đầu đọc các thuộc tính từ nó bằng cách sử dụng kiểu java như hình bên dưới:

```
@Service
public class ContactService {

    @Autowired
    EazySchoolProps eazySchoolProps;

    public Page<Contact> findMsgsWithOpenStatus(int pageNum, String sortField, String sortDir) {
        int pageSize = eazySchoolProps.getPageSize();
        if(null!=eazySchoolProps.getContact() && null!=eazySchoolProps.getContact().get("pageSize")){
            pageSize = Integer.parseInt(eazySchoolProps.getContact().get("pageSize").trim());
        }
    }
}
```

5. Reading properties with @PropertySource

Trong Java Spring, @PropertySource là một chú thích được sử dụng để chỉ định các tệp tin cấu hình chứa các giá trị thuộc tính. Nó được sử dụng trong cấu hình của Spring để chỉ định các nguồn tệp tin cấu hình mà Spring sẽ đọc để lấy giá trị thuộc tính.

Ví dụ sử dụng @PropertySource:

```
@Configuration
@PropertySource("classpath:my.properties")
public class AppConfig {
    // Cấu hình khác của ứng dụng
}
```

Trong ví dụ trên, chúng ta sử dụng chú thích @PropertySource trên lớp cấu hình AppConfig để chỉ định tệp tin my.properties là tệp tin cấu hình được sử dụng. Spring sẽ đọc các giá trị thuộc tính từ tệp tin này và có thể sử dụng chúng trong các thành phần khác của ứng dụng.

ignoreResourceNotFound: Đây là một thuộc tính boolean mặc định được đặt là false. Nếu bạn đặt ignoreResourceNotFound = true trong chú thích @PropertySource, Spring sẽ không phát ra lỗi nếu không tìm thấy tệp tin cấu hình. Thay vào đó, nó sẽ bỏ qua lỗi và không đọc giá trị thuộc tính từ tệp tin đó.

```
@PropertySource(value = "classpath:my.properties", ignoreResourceNotFound = true)
```

Multiple PropertySource: Bạn có thể sử dụng nhiều chú thích @PropertySource trên cùng một lớp cấu hình để chỉ định nhiều tệp tin cấu hình. Trong trường hợp này, các tệp tin cấu hình được xem như một chuỗi các nguồn, và giá trị thuộc tính sẽ được tìm kiếm theo thứ tự từ trái qua phải.

```
@PropertySource("classpath:config1.properties")
@PropertySource("classpath:config2.properties")
```

Trong ví dụ trên, Spring sẽ tìm kiếm giá trị thuộc tính từ config1.properties trước, sau đó từ config2.properties. Nếu hai tệp tin cấu hình có cùng một thuộc tính, giá trị trong tệp tin được khai báo sau sẽ được ưu tiên.

Với sử dụng @PropertySource và các thuộc tính liên quan, bạn có thể chỉ định các tệp tin cấu hình và đọc giá trị thuộc tính từ chúng trong Java Spring.

6. Introduction to Profiles in Spring(example49)

Trong Spring Framework, profile là một cách để cấu hình và quản lý các biến môi trường (environment variables), thuộc tính (properties), và các thành phần khác của ứng dụng dựa trên môi trường chạy (runtime environment) hoặc cấu hình được chỉ định.

Profile cho phép bạn định nghĩa các tập hợp cấu hình khác nhau dựa trên các yếu tố như môi trường chạy, giai đoạn phát triển, vị trí triển khai, v.v. Mỗi profile có thể chứa các tệp cấu hình và các giá trị thuộc tính riêng, và bạn có thể kích hoạt hoặc vô hiệu hóa profile tương ứng để áp dụng các cấu hình tương ứng.

Để sử dụng profile trong Spring, bạn có thể thực hiện các bước sau:

1. Định nghĩa các tệp cấu hình cho từng profile: Tạo các tệp cấu hình có tên đặc biệt, ví dụ: application-dev.properties, application-prod.properties, application-test.properties, v.v. Mỗi tệp cấu hình sẽ chứa các giá trị thuộc tính tương ứng với profile đó.
2. Kích hoạt profile trong ứng dụng: Có nhiều cách để kích hoạt profile trong Spring:
 - Trong tệp cấu hình application.properties hoặc application.yml, bạn có thể sử dụng thuộc tính spring.profiles.active để chỉ định profile muốn kích hoạt. Ví dụ: spring.profiles.active=dev.
 - Bạn có thể sử dụng cờ --spring.profiles.active khi chạy ứng dụng từ dòng lệnh: java -jar myapp.jar --spring.profiles.active=dev.
 - Trong môi trường phát triển, bạn có thể sử dụng các lớp cấu hình đặc biệt để kích hoạt profile mà không cần sửa đổi tệp cấu hình.
3. Sử dụng giá trị thuộc tính trong profile: Để sử dụng giá trị thuộc tính trong profile, bạn có thể sử dụng chú thích @Value hoặc @ConfigurationProperties trong các lớp và thành phần Spring. Các giá trị thuộc tính sẽ được đọc từ tệp cấu hình tương ứng với profile đang được kích hoạt.

```
@Configuration
@PropertySource("classpath:application-dev.properties")
public class AppConfig {
    @Value("${my.property}")
    private String myProperty;
}
```

Trong ví dụ trên, chúng ta đã định nghĩa một lớp cấu hình AppConfig và sử dụng chú thích @PropertySource để chỉ định tệp cấu hình application-dev.properties cho profile "dev". Sau đó, chúng ta sử dụng chú thích @Value để đọc giá trị thuộc tính my.property từ tệp cấu hình tương ứng.

Qua đó, bạn có thể sử dụng profile trong Spring để quản lý các cấu hình và giá trị thuộc tính dựa trên môi trường chạy hoặc cấu hình được chỉ định.

7. Hướng dẫn sử dụng Spring Profiles

a) Tạo file config

Để sử dụng, tạo file config tại thư mục `resources` trong project. Mặc định Spring sẽ nhận các file có tên như sau:

```
application.properties  
application.yml  
application-{profile-name}.yml // .properties
```

Ví dụ có 2 môi trường là `local` và `aws`, thì sẽ tạo ra các file như thế này:

```
application.yml  
application-local.yml  
application-aws.yml  
application-common.yml
```

- `application` là file config chính khai báo các environment.
- `application-local` chỉ sử dụng khi chạy chương trình ở local
- `application-aws` chỉ sử dụng khi chạy ở AWS
- `application-common` là những config dùng chung, môi trường nào cũng cần.

Bây giờ, sẽ khai báo trong từng file như sau:

```
application.yml
```

```
#application.yml  
---  
spring.profiles: local  
spring.profiles.include: common, local  
---  
spring.profiles: aws
```

```
spring.profiles.include: common, aws
```

```
---
```

```
application-aws.yml
```

```
spring:  
  datasource:  
    username: xxx  
    password: xxx  
    url: jdbc:mysql://10.127.24.12:2030/news?useSSL=false&characterEncoding=UTF-8
```

```
application-local.yml
```

```
spring:  
  datasource:  
    username: root  
    password:  
    url: jdbc:mysql://localhost:3306/news?useSSL=false&characterEncoding=UTF-8  
  
  logging:  
    level:  
      org:  
        hibernate:  
          SQL: debug
```

```
application-common.yml
```

```
spring:
```

```
jpa:  
  
    properties:  
  
        hibernate:  
  
            jdbc:  
  
                batch_size: 50  
  
                batch_versioned_data: true  
  
        hibernate:  
  
            ddl-auto: none
```

b) Kích hoạt config

Để sử dụng một Profiles có các cách sau:

#1: Sử dụng `spring.profiles.active` trong file `application.properties` hoặc `application.yml`

```
spring.profiles.active=aws
```

#2: Active trong code, trước khi chạy chương trình.

```
@Configuration  
  
public class ApplicationInitializer  
  
implements WebApplicationInitializer {  
  
    @Override  
  
    public void onStartup(ServletContext servletContext) throws ServletException {  
  
        servletContext.setInitParameter(  
            "spring.profiles.active", "aws");  
  
    }  
  
}
```

hoặc

```
@Autowired  
private ConfigurableEnvironment env;  
...  
env.setActiveProfiles("aws");
```

hoặc

```
SpringApplication application = new SpringApplication(SpringBootProfilesApplication.class);  
ConfigurableEnvironment environment = new StandardEnvironment();  
environment.setActiveProfiles("aws");  
application.setEnvironment(environment);  
application.run(args);
```

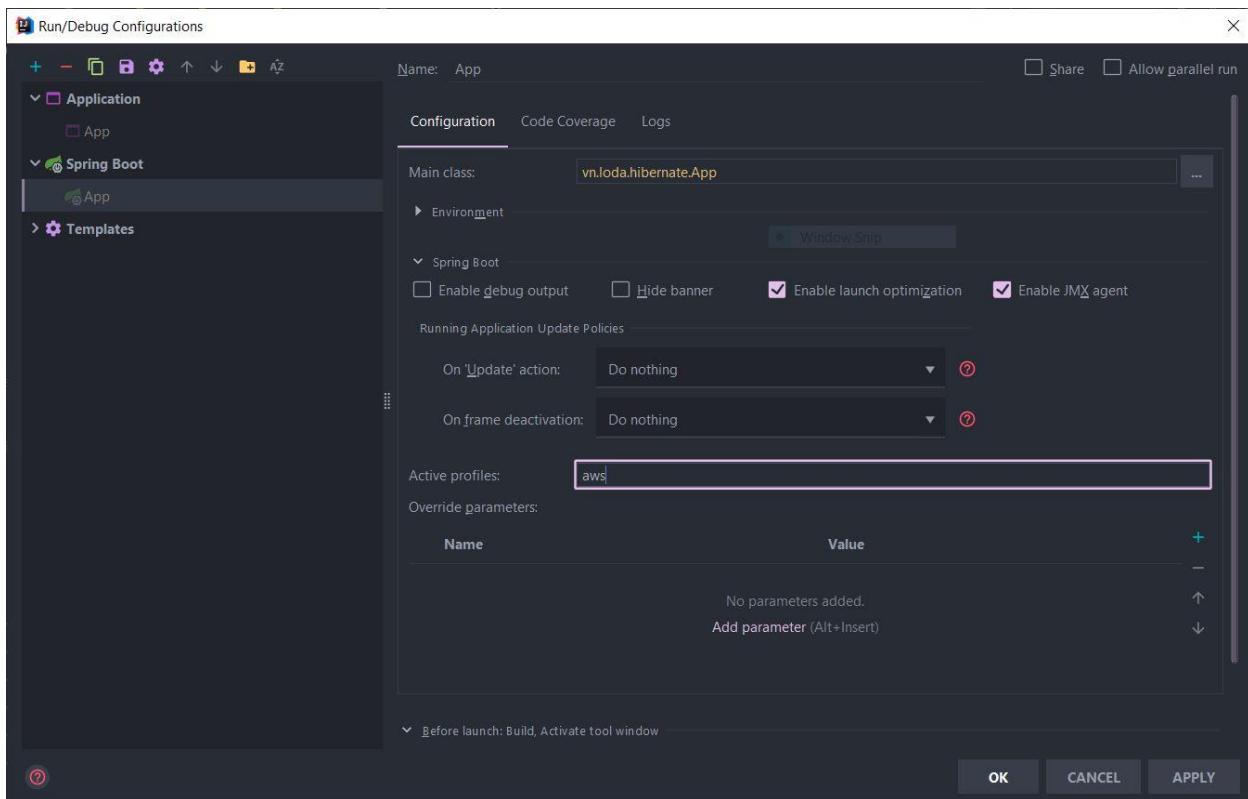
#3: Sử dụng JVM System Parameter (nên dùng)

```
-Dspring.profiles.active=aws
```

#4: Environment Variable (Unix) (nên dùng)

```
export SPRING_PROFILES_ACTIVE=aws
```

Nếu ai sử dụng IntelliJ IDEA thì có thể config ngay trong IDE như thế này, mỗi lần chạy nó tự active cho mình.



c) Cách sử dụng @Profile

Khi đã có Profile rồi, ngoài các biến toàn cục được thay đổi theo môi trường, cũng có thể toàn quyền quyết định xem trong code rằng Bean hay Class nào sẽ được quyền chạy ở môi trường nào. Bằng cách sử dụng annotation `@Profile`

```
@Component
@Profile("local")
public class LocalDatasourceConfig
```

Ngoài ra bạn có thể sử dụng toàn tử logic ở đây, ví dụ:

```
// Bean này Spring chỉ khởi tạo và quản lý khi môi trường là những môi trường không phải là 'local'
@Component
@Profile("!local")
public class LocalDatasourceConfig
```

Section 21: Deep Dive on Spring Boot Actuator & Spring Boot Admin

Actuator là một module mạnh mẽ và hữu ích trong Spring Boot, cung cấp các chức năng giám sát và quản lý ứng dụng. Nó cho phép bạn khám phá, giám sát và quản lý ứng dụng Spring Boot một cách dễ dàng thông qua các API REST và giao diện người dùng.

Một số tính năng quan trọng của Actuator bao gồm:

1. Health: Actuator cung cấp điểm cuối /health để kiểm tra trạng thái sức khỏe của ứng dụng. Bằng cách truy cập API /health, bạn có thể kiểm tra xem ứng dụng có đang hoạt động, có vấn đề gì hay không và các chi tiết khác về trạng thái sức khỏe.
2. Metrics: Actuator cung cấp thông tin liên quan đến các chỉ số và số liệu thống kê về hoạt động của ứng dụng. Bạn có thể truy cập API /metrics để thu thập dữ liệu về tài nguyên, tình trạng, hiệu suất và các thông số khác của ứng dụng.
3. Tracing: Actuator cho phép bạn theo dõi các yêu cầu HTTP và xem thông tin chi tiết về các hoạt động theo dõi. Bằng cách kích hoạt tính năng "tracing", bạn có thể xem các đoạn mã (spans) và liên kết chúng lại thành một chuỗi (trace) để theo dõi quá trình thực hiện các yêu cầu trong ứng dụng.
4. Auditing: Actuator cung cấp thông tin về các sự kiện quan trọng xảy ra trong ứng dụng, ví dụ như các tác động CRUD lên cơ sở dữ liệu. Bằng cách sử dụng tính năng "auditing", bạn có thể xem lịch sử các sự kiện quan trọng trong ứng dụng.
5. Info: Actuator cho phép bạn tạo các thông tin tùy chỉnh về ứng dụng và cung cấp chúng thông qua API /info. Bạn có thể thêm thông tin như phiên bản ứng dụng, môi trường triển khai, thông tin liên lạc, v.v.

Để sử dụng Actuator trong ứng dụng Spring Boot, bạn chỉ cần thêm spring-boot-starter-actuator vào phần dependencies của file pom.xml hoặc build.gradle. Sau đó, khi ứng dụng được chạy, các điểm cuối Actuator sẽ tự động được cung cấp và sẵn sàng để sử dụng.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Truy cập vào API <http://localhost:8080/actuator>

Bạn cũng có thể tùy chỉnh các điểm cuối Actuator bằng cách cấu hình trong file application.properties hoặc application.yml. Ví dụ, để bật tính năng tracing, bạn có thể đặt management.tracing.enabled=true.

Vì lý do bảo mật, nên ở phiên bản 2.x của Spring, theo mặc định, chỉ có một số API được enable, để enable hết tất cả các API mà Spring Boot Actuator cung cấp, chúng ta có thể thêm config trong file cấu hình như sau:

```
management.endpoints.web.exposure.include=*
```

API paths provided by Actuator

HTTP method	Path	Description
GET	/auditevents	Produces a report of any audit events that have been fired.
GET	/conditions	Produces a report of autoconfiguration conditions that either passed or failed, leading to the beans created in the application context.
GET	/configprops	Describes all configuration properties along with the current values.
GET	/beans	Describes all the beans in the Spring application context.
GET, POST, DELETE	/env	Produces a report of all property sources and their properties available to the Spring application.
GET	/env/{toMatch}	Describes the value of a single environment property.
GET	/heapdump	Downloads the heap dump.
GET	/health	Returns the aggregate health of the application and (possibly) the health of external dependent applications.
GET	/httptrace	Produces a trace of the most recent 100 requests.
GET	/info	Returns any developer-defined information about the application.
GET	/loggers	Produces a list of packages in the application along with their configured and effective logging levels.

API paths provided by Actuator

HTTP method	Path	Description
GET, POST	/loggers/{name}	Returns the configured and effective logging level of a given logger. The effective logging level can be set with a POST request.
GET	/mappings	Produces a report of all HTTP mappings and their corresponding handler methods.
GET	/scheduledtasks	Lists all scheduled tasks.
GET	/threaddump	Returns a report of all application threads.
GET	/metrics	Returns a list of all metrics categories.
GET	/metrics/{name}	Returns a multidimensional set of values for a given metric.