## **Tuần 3: Cơ bản về Spring Boot**

### **Ngày 1: Cơ bản về Spring Boot**

#### **Giới thiệu về Spring Framework: lợi ích, lịch sử phát triển**

##### **Tổng quan**

* Spring ra đời vào **năm 2003**. Johnson, cùng với Juergen Hoeller và Yann Caroff, đã phát triển Spring thành một **framework mã nguồn mở**.
* Họ đã tạo ra Spring để giúp phát triển Java phía máy chủ dễ dàng hơn và cho phép các nhóm phát triển tạo ứng dụng của họ nhanh hơn.
* Spring là một Framework phát triển ứng dụng Java rất phổ biến và được sử dụng bởi nhiều lập trình viên. Với việc Spring đứng thứ tư trong 'Framework được yêu thích nhất' trong cuộc khảo sát dành cho nhà phát triển năm 2019 của Stack Overflow.

##### **Kiến trúc tổng thể của Spring Framework**

* **Test**:
* Nhóm này cung cấp khả năng hỗ trợ kiểm thử với **JUnit, TestNG**.
* **Spring Core Container**:
* Bao gồm các module **spring core**, **beans**, **context**, **SpEL**(Expression Language).
* **Spring core, beans** cung cấp các tính năng như **dependency injection** và **inversion of control**.
* **Spring context** hỗ trợ đa ngôn ngữ, các tính năng **Java EE** như **EJB**, **JMX**.
* **SpEL**: được mở rộng từ **Expression Language của JSP**. Cung cấp hỗ trợ việc getting/setting giá trị, các method cải tiến cho phép truy cập collection, index, toán tử logic…
* **Aop, Aspect, và Instrumentation**:
* Hỗ trợ cài đặt lập trình hướng khía cạnh (**Aspect of Programming**), tích hợp **AspectJ**.
* **Data Access / Integration:**
* Nhóm này bao gồm **JDBC**, **ORM**, **OXM**, **JMS** và module **transaction**. Những module này cung cấp khả năng giao tiếp với **cơ sở dữ liệu**.
* **Web:**
* Hay còn gọi là **Spring MVC**, gồm **Web**, **Web-Servlet** hỗ trợ cho việc tạo ứng dụng **Web**.
* **Các thành phần khác**:
* **Spring MVC:**
* Thiết kế cho việc xây dựng các ứng dụng nền tảng web.
* **Spring Security:**
* Cung cấp cơ chế **xác thực** (**Authentication**) và **phân quyền** (**Authorization**) cho ứng dụng.
* **Spring Boot:**
* Một **framework** giúp phát triển cũng như chạy ứng dụng nhanh chóng.
* **Spring Batch:**
* Hỗ trợ tạo các **lịch trình (scheduling)** và **tiến trình (processing)** cho các công việc **xử lý đồng thời**.

##### **Lợi ích**

* **Đơn giản**: Spring cho phép làm việc với các lớp POJO, không sử dụng EJB, ứng dụng, luồng chạy, cấu hình.
* **Hỗ trợ sử dụng nhiều công nghệ** khác nhau: ORM Framework, Logging Framework, JEE ,...
* **Module Web Spring** theo mô hình MVC cung cấp đầy đủ tính năng, giúp thay thế   
  framework khác như Struts.
* Nó giúp tạo ra các ứng dụng Java **hiệu suất cao, dễ kiểm thử** và có khả năng **tái sử dụng** mã code.
* Spring là một **dự án mã nguồn mở**, có cộng đồng người dùng lớn và được chia sẻ miễn phí.
* Spring có **kích thước nhỏ gọn** và dễ sử dụng.
* **Đa nền tảng** có thể được sử dụng để phát triển các ứng dụng Java trên nhiều nền tảng như desktop, di động và web

#### **Giới thiệu về Spring Boot: lợi ích, lịch sử phát triển**

##### **Tổng quan**

* Vào **năm 2012**, Spring Boot ra đời bởi Phil Webb như một cách để làm cho việc thiết kế và phát triển các ứng dụng web bằng Spring dễ tiếp cận hơn.
* Một trong số các module của Spring framework chuyên cung cấp các tính năng RAD (Rapid Application Development) cho phép tạo ra và phát triển các ứng dụng độc lập dựa trên Spring một cách nhanh chóng.

##### **Lợi ích**

* Hội tụ **đầy đủ các tính năng** của Spring framework.
* **Đơn giản** hóa cấu hình và xây dựng được các ứng dụng độc lập dựa trên Spring một cách nhanh chóng.
* **Dễ dàng deploy** lên môi trường production mà không cần thiết phải tải file WAR.
* Cung cấp **nhiều plugin, số liệu, cấu hình ứng dụng** từ bên ngoài.
* Giúp nhúng trực tiếp Tomcat, Jetty hoặc Undertow.
* Mục đích là để **giảm LOC**.
* Dễ dàng khởi chạy hơn.
* Tùy chỉnh và quản lý dễ dàng hơn.

#### **Các khái niệm cơ bản: Spring Container, Dependency Injection (DI), Inversion of Control (IoC)**

##### **Spring Container**

* **Container Spring** là cốt lõi của **Spring Framework**. **Container** sẽ tạo các đối tượng, kết nối chúng lại với nhau, cấu hình chúng và **quản lý** toàn bộ **vòng đời** của chúng từ khi tạo cho đến khi hủy. Các đối tượng này được gọi là **Spring Beans**
* **Container Spring** sử dụng **DI** để quản lý các thành phần tạo nên ứng dụng.
* **Container** nhận được hướng dẫn về các đối tượng cần khởi tạo, cấu hình và lắp ráp bằng các dữ liệu cấu hình được cung cấp. Dữ liệu cấu hình có thể được biểu diễn bằng **XML, annotations** **Java** hoặc **mã Java**. Sau đó, container sử dụng các **lớp Java POJO** và **dữ liệu cấu hình** để tạo ra một hệ thống hoặc ứng dụng được cấu hình và thực thi đầy đủ.
* Spring cung cấp hai loại container riêng biệt:

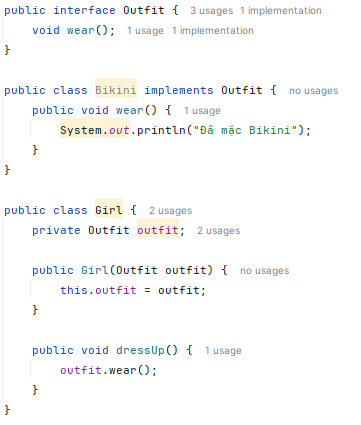
| **Spring BeanFactory Container** | **Spring ApplicationContext Container** |
| --- | --- |
| Container đơn giản nhất cung cấp hỗ trợ cơ bản cho DI và được BeanFactory và các giao diện liên quan, chẳng hạn như BeanFactoryAware, InitializingBean, DisposableBean, vẫn có trong Spring nhằm mục đích tương thích ngược với nhiều khuôn khổ của bên thứ ba tích hợp với Spring. | Container này bổ sung thêm nhiều chức năng dành riêng cho doanh nghiệp như khả năng giải quyết tin nhắn văn bản từ tệp thuộc tính và khả năng xuất bản các sự kiện ứng dụng cho người nghe sự kiện quan tâm. |
| Định nghĩa bởi giao diện **org.springframework.beans.factory.BeanFactory** | Định nghĩa bởi giao diện **org.springframework.context.ApplicationContext** |

* Lưu ý:
* **Container ApplicationContext** bao gồm tất cả các chức năng của container BeanFactory , do đó, nó thường được khuyến nghị hơn **BeanFactory** .
* **BeanFactory** vẫn có thể được sử dụng cho các ứng dụng nhẹ như thiết bị di động hoặc các ứng dụng dựa trên applet

##### **Dependency Injection**

* Đây là một cách để hiện thực Inversion of Control Pattern (hoặc một design pattern riêng). Các module phụ thuộc (dependency) sẽ được inject vào module cấp cao.
* Mục đích của sử dụng DI:
* Các module không giao tiếp trực tiếp với nhau, mà thông qua interface. Module cấp thấp sẽ implement interface, module cấp cao sẽ gọi module cấp thấp thông qua interface.
* Việc khởi tạo các module cấp thấp sẽ do DI Container thực hiện.
* Việc Module nào gắn với interface nào sẽ được config trong code hoặc trong file XML.
* DI được dùng để làm giảm sự phụ thuộc giữa các module, dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.
* Có **3 dạng Dependency Injection**:
* **Constructor Injection**: Các dependency sẽ được container **truyền vào (inject vào)** 1 class thông qua constructor của class đó. Đây là cách thông dụng nhất.
* **Setter Injection**: Các dependency sẽ được truyền vào 1 class thông qua các hàm Setter.
* **Interface Injection**: Class cần inject sẽ implement 1 interface. Interface này chứa 1 hàm tên Inject. Container sẽ injection dependency vào 1 class thông qua việc gọi hàm Inject của interface đó. Đây là cách rườm rà và ít được sử dụng nhất.

| **ƯU ĐIỂM** | **KHUYẾT ĐIỂM** |
| --- | --- |
| Giảm sự kết dính giữa các module | Khái niệm DI khá “khó tiêu”, các developer mới sẽ gặp khó khăn khi học |
| Code dễ bảo trì, dễ thay thế module | Sử dụng interface nên đôi khi sẽ khó debug, do không biết chính xác module nào được gọi |
| Rất dễ test và viết Unit Test | Các object được khởi tạo toàn bộ ngay từ đầu, có thể làm giảm performance |
| Dễ dàng thấy quan hệ giữa các module (Vì các dependency đều được inject vào constructor) | Làm tăng độ phức tạp của code |



* Lớp Girl không tạo ra một Outfit mà nó cần. Thay vào đó, nó chấp nhận một Outfit thông qua constructor của nó. Điều này làm giảm sự phụ thuộc của Girl vào một cụ thể Outfit, tạo điều kiện cho tính linh hoạt và tùy biến.
* **DI** giúp **giảm sự phụ thuộc** giữa các thành phần trong ứng dụng, tạo tính **linh hoạt** và **tùy biến**.

##### **Inversion of Control**

* là một nguyên tắc lập trình, trong đó luồng điều khiển trong ứng dụng không được quyết định bởi ứng dụng mà được quyết định bởi một **framework** hoặc **container** bên ngoài.
* **IoC** thường đi kèm với **DI**, nơi các **dependency** được quản lý và cung cấp bởi một **framework** hoặc **container**. **Framework** sẽ quản lý việc tạo và cung cấp các phụ thuộc (**dependency**).
* **IoC** là một **design pattern** để hiện thực hóa hay implement nguyên lý thiết kế **Dependency Inversion**:
* Các module, class cấp cao (high-level) không nên phụ thuộc vào module, class cấp thấp hơn (low-level) mà nên phụ thuộc (giao tiếp) với nhau thông qua một abstraction (Interface).
* Abstraction không nên phụ thuộc vào chi tiết. Chi tiết nên phụ thuộc vào abstraction.
* Trong ví dụ này, **Spring Framework** quản lý việc tạo ra đối tượng Girl và cung cấp Outfit thông qua **DI**, không cần tự tạo đối tượng Girl, Spring sẽ làm điều đó và cung cấp Outfit.
* **IoC** cho phép **framework** quản lý việc **tạo và cung cấp** **dependency**, tạo điều kiện cho việc **quản lý phức tạp hóa ứng dụng**.

### **Ngày 2: Cấu hình và khởi chạy ứng dụng**

#### **Cấu hình ứng dụng Spring Boot (application.properties/yml)**

##### **Spring Initializr**

* Spring Boot so với Spring Framework là tối ưu hóa quá trình config dự án. Spring Initializr là yếu tố quan trọng của quá trình tối ưu.
* Spring Initializr có địa chỉ tại <https://start.spring.io/>. Đối với IntelliJ Ultimate thì công cụ này được tích hợp trong IDE.
* Các thông tin project:
* **Loại project**: lựa chọn package manager cho project (Maven, Gradle).
* **Ngôn ngữ**: ngôn ngữ lập trình sử dụng.
* **Spring Boot**: phiên bản của Spring Boot sử dụng.
* **Packaging**: lựa chọn loại file build ra, JAR hoặc WAR.
* **Java**: phiên bản Java.
* **Group**: thông tin về nhóm.
* **Artifact**: thông tin về ID.
* **Name**: Tên project.
* **Description**: mô tả.
* **Package name**: tên của package sau khi khởi tạo.

##### **Lựa chọn dependencies cho project**

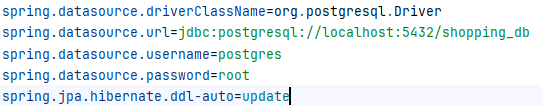
* Mục đích để **lựa chọn các thư viện** (lib).
* Các **dependencies** thường có:
* **Lombok**: hỗ trợ code ngắn gọn bằng cách rút gọn một số boiler-plate code qua **annotation như Getter, Setter, Constructor**, ….
* **Spring Data JPA**: xử lý data và làm việc với Database qua **JPA và Hibernate.**
* **Spring Boot DevTools**: tính năng hỗ trợ LiveReload **giúp khởi động lại ứng dụng nhanh hơn**, giảm thời gian chờ sau mỗi lần chỉnh sửa source code.

##### **Hoàn thành**

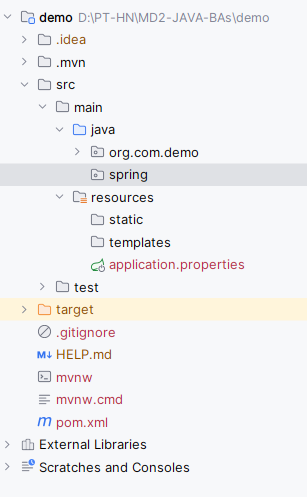
* Sau khi đã cung cấp các thông tin của project và lựa chọn dependencies cần thiết, nhất nút Generate.

##### **Cấu hình và kết nối cơ sở dữ liệu (H2, MySQL, PostgreSQL)**

* Cấu hình file **pom.xml**:
* Thêm thẻ dependency driver của PostgreSQL vào file.
* Chỉnh sửa file application.properties:



#### **Cấu trúc dự án Spring Boot**

* Thư mục gốc chứa các file như pom.xml (của Maven), build.gradle và các file khác như .gitignore,... dùng để cấu hình dự án.
* Thư mục .mvn hoặc .gradle là thư mục riêng của Maven và Gradle, đừng nên đụng tới hay exclude nó ra khỏi source code.
* Code được chứa trong thư mục src.
* Thư mục build ra chứa các file class, file JAR. Với Maven là target còn Gradle là build.
* Tên package chính được đặt dạng ngược với tên miền. Ví dụ như domain.com thì đặt thành com.domain. Cộng thêm tên project nữa.
* Có các package con, mỗi package đại diện cho các class thuộc layer cụ thể (ví dụ như service, controller,...)
* Thư mục resources chứa các tài nguyên của ứng dụng như hình ảnh, static file, properties file,...
* Ngoài ra còn có src/test dùng để chứa các test class, dùng cho unit test.

#### **Cách khởi chạy và các chế độ chạy ứng dụng (dev, test, prod)**

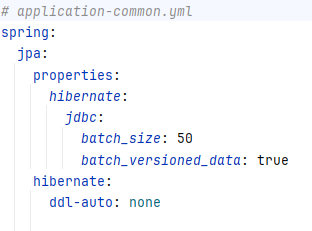
* Spring Profiles là một core feature trong Spring Framework cho phép cấu hình ứng dụng tùy theo môi trường.

##### **Tạo file config**

* Để sử dụng cần tạo file config tại thư mục resource trong project. Mặc định Spring sẽ nhận các file có tên:
* application.properties.
* application.yml.
* application-{profile-name}.yml
* Có 3 môi trường là dev, test, prod thì sẽ tạo các file:
* application.yml.
* application-dev.yml.
* application-test.yml.
* application-prod.yml.
* application-common.yml.
* Khai báo trong từng file:







##### **Kích hoạt config**

* Sử dụng spring.profiles.active trong file application.properties hoặc application.yml.



* Environment Variable (Unix) (nên dùng)
* Sử dụng Intellij IDEA thì có thể config ngay trong IDE như thế này, mỗi lần chạy nó tự active cho mình.

##### **Cách sử dụng profile**

* Khi đã có Profile rồi, ngoài các biến toàn cục được thay đổi theo môi trường, bạn 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**.
* Có thể sử dụng toán tử logic.

#### **Các Annotation cơ bản: @SpringBootApplication, @RestController, @RequestMapping, @GetMapping, @PostMapping**

##### **@SpringBootApplication**

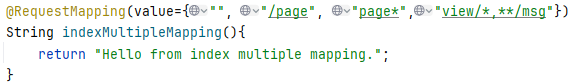
* Sử dụng annotation cho main class của 1 ứng dụng Spring Boot.
* **@SpringBootApplication** đóng gói 3 annotation **@Configuration**, **@EnableAutoConfiguration** và **@ComponentScan bên trong**:
* **@EnableAutoConfiguration** cho phép SpringBoot tự động tìm kiếm auto-configuration bean() trong classpath và tự động áp dụng. Phải sử dụng với @Configuration.

##### **@RestController**

* là một chú thích quan trọng trong Spring Boot. Khác với **@Controller**, nó không trả về một template mà thay vào đó trả về dữ liệu dưới dạng **JSON**.
* Các đối tượng được trả về sẽ tự động được Spring Boot chuyển đổi thành JSON. Bạn có thể trả về nhiều loại đối tượng như List, Map,.. và nó sẽ được chuyển đổi một cách tự động. Spring Boot sử dụng **Jackson converter** mặc định để thực hiện việc này.
* Chú ý:
* Có thể tùy chỉnh kiểu dữ liệu trả về bằng cách sử dụng **ResponseEntity**

##### @**RequestMapping**

* Để định cấu hình ánh xạ các web request, chúng ta sử dụng annotation @RequestMapping.
* Annotation @RequestMapping cấp lớp ánh xạ một đường dẫn hoặc mẫu request cụ thể lên controller. Sau đó, ta có thể áp dụng các annotation @RequestMapping cấp phương thức bổ sung để làm cho ánh xạ cụ thể hơn đối với các phương thức xử lý.
* **@RequestMapping** với nhiều URI

****

* Annotation **@RequestMapping** có khả năng xử lý các phương thức yêu cầu HTTP, chẳng hạn như **GET, PUT, POST, DELETE và PATCH**. Theo **mặc định**, tất cả các yêu cầu được coi là loại **HTTP GET**.

##### **@GetMapping**

* Cách tiếp cận mới giúp rút ngắn mã code cũng như chú trọng hơn vào endpoint method như GET, POST, PUT, DELETE.
* Chú thích **@GetMapping** là phiên bản @RequestMapping chú thích được biên soạn có chức năng như một phím tắt cho **@RequestMapping(method = RequestMethod.GET)**.

##### **@PostMapping**

* Đây **@PostMapping** là phiên bản **@RequestMapping** chú thích chuyên biệt đóng vai trò là phím tắt cho **@RequestMapping(method = RequestMethod.POST).**
* **Thuộc tính** của annotation:
* **path hoặc value**: chỉ định URI ánh xạ.
* **consumes**: bao gồm một hoặc nhiều loại, một trong số đó phải khớp với Content-Type tiêu đề yêu cầu.
* **produces**: bao gồm một hoặc nhiều loại, một trong số đó phải được chọn.
* **headers**: chỉ định các tiêu đề của yêu cầu được ánh xạ.
* **name**: gán tên cho mapping.
* **params**: chỉ định các tham số của yêu cầu được ánh xạ.

#### **Khái niệm Layered Architecture: Controller, Service, Repository**

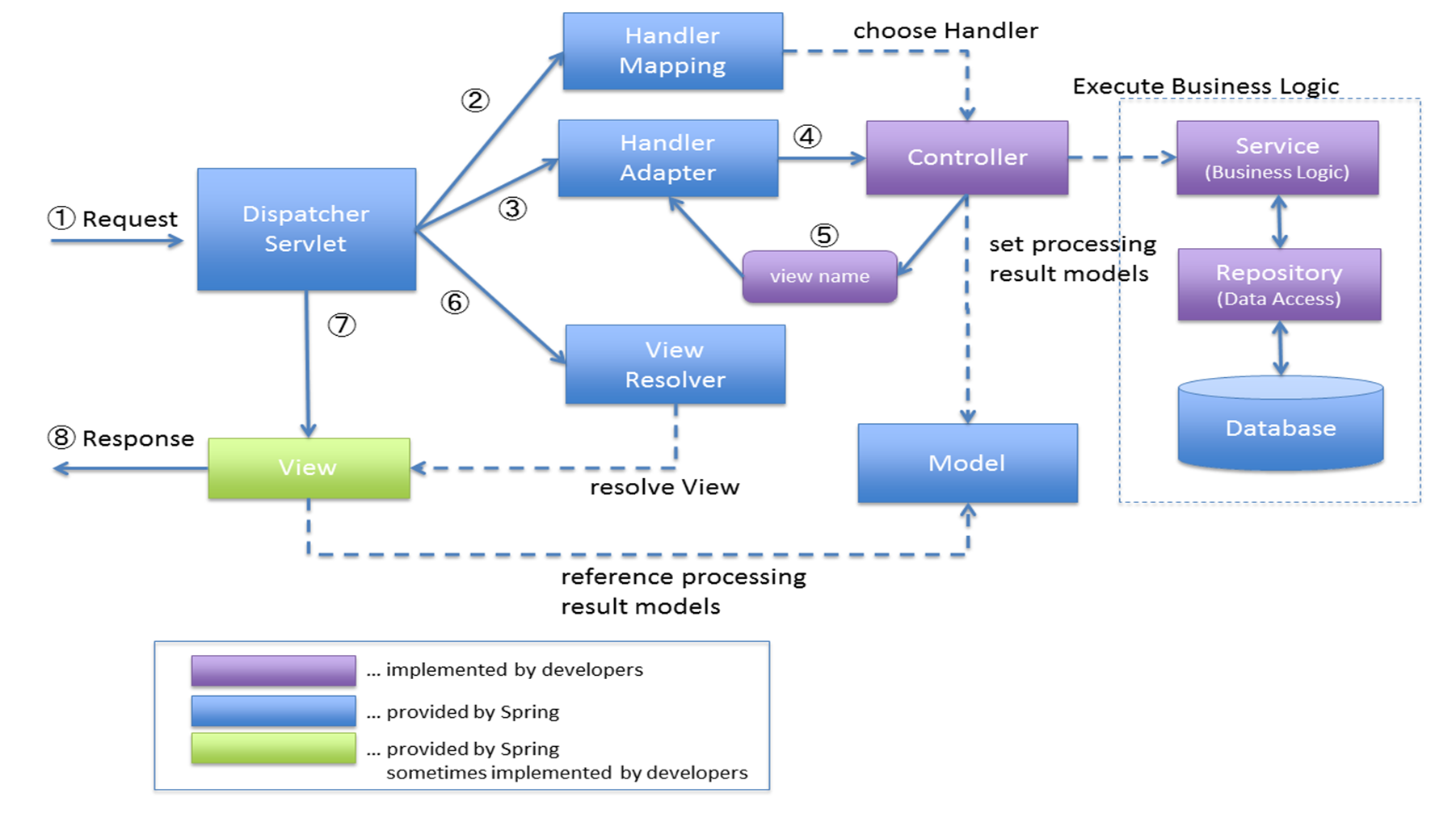
##### **Mô hình 3 lớp**

* **Presentation layer:** tầng này tương tác với người dùng, bằng View, Controller (trong MVC) hoặc API (nếu có).
* **Business logic layer:** Chứa toàn bộ logic của chương trình, đa số code nằm ở đây
* **Data access layer:** Tương tác với database, trả về kết quả cho tầng business logic
* **Trong Spring Boot, thì có một số thành phần đại diện cho từng lớp:**
* **Service**: chứa các business logic code.
* **Repository**: đại diện cho tầng data access.

##### **Mô hình MVC**

* **Model:** các cấu trúc dữ liệu của toàn chương trình, có thể đại diện cho trạng thái của ứng dụng
* **View**: lớp giao diện, dùng để hiển thị dữ liệu ra cho user xem và tương tác
* **Controller:** kết nối giữa **Model** và **View**, điều khiển dòng dữ liệu
* Dữ liệu từ **Model** qua **Controller** sau đó được gửi cho **View** hiển thị ra. Và ngược lại, khi có yêu cầu mới từ **View**, thì sẽ qua **Controller** thực hiện thay đổi dữ liệu của **Model**.
* Lưu ý:
* **MVC** chỉ mô tả luồng đi của dữ liệu, nó không nói rõ như code đặt ở đâu (ở Model, View hay Controller).
* Đối với ứng dụng hoàn chỉnh như Spring Boot thì cần kết hợp cả mô hình **MVC và 3-tier** lại với nhau.

##### **Kết hợp MVC và 3-tier**

* **Controller**: trả về View (có chứa data sẵn, dạng trang HTML), hoặc Model thể hiện dưới dạng API cho View (View viết riêng bằng React, Vue, hoặc Angular).
* **Service**: chứa các code tính toán, xử lý. Khi Controller yêu cầu, thì Service tương ứng sẽ tiếp nhận và cho ra dữ liệu trả cho Controller (trả về Model). Controller sẽ gửi về View như trên.
* **Repository**: Service còn có thể tương tác với service khác, hoặc dùng Repository để gọi DB. Repository trực tiếp tương tác, đọc ghi dữ liệu trong DB và trả cho service.
* **View** có 2 loại:
* **View dạng 1** là dạng truyền thống là trả về 1 cục HTML có data rồi. Lúc này Controller sẽ pass dữ liệu vào View và return về (Spring MVC có JSP hoặc template engine như Thymeleaf làm điều đó).
* **View dạng 2** là dạng View tách riêng (không thuộc về project Spring boot). Thường có trong các hệ thống dùng API. View sẽ được viết riêng bằng React, Angular,… Controller sẽ đưa dữ liệu Model thông qua API cho View, và cũng nhận lại các yêu cầu qua API luôn.

### **Ngày 3: Spring Data JPA**

#### **Giới thiệu về JPA (Java Persistence API) và Spring Data JPA**

##### **JPA**

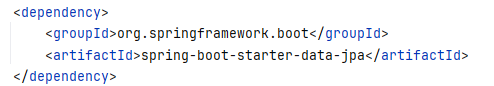
* Sau này được đổi tên thành **Jakarta Persistence API**.
* **JPA** giải quyết các vấn đề trong quản lý quan hệ giữa các thực thể, giúp map những object **Java (POJO – Plain Old Java Object)** thành những **bảng trong database**.
* Java Persistence API, Java là tên ngôn ngữ, API ám chỉ rằng đây là một framework cung cấp các api, các logic mà ta dễ dàng sử dụng lại giúp tăng tốc độ phát triển ứng dụng
* Object persistence nghĩa là các thực thể có thể “sống lâu” hơn các chương trình, có thể lưu trữ trong các nơi quản lý dữ liệu và có thể khởi tạo lại trong chương trình vào thời điểm nào đó.
* Vai trò:
* **Viết code ít hơn**: sử dụng các đối tượng Java thay vì phải viết các câu truy vấn SQL hoặc sử dụng JDBC trực tiếp. Điều này làm cho code trở nên ngắn gọn hơn và dễ đọc hơn, giảm thiểu tối đa boilerplate code.
* **Dễ bảo trì**: việc thay đổi cấu trúc cơ sở dữ liệu trở nên dễ dàng hơn. Chúng ta chỉ cần thay đổi định nghĩa của đối tượng Java và JPA sẽ tự động áp dụng các thay đổi này vào cơ sở dữ liệu tương ứng.
* **Hiệu suất cao**: tối ưu hóa việc truy xuất và lưu trữ dữ liệu trong cơ sở dữ liệu. Nó sử dụng các kỹ thuật như lazy loading và caching để giảm thiểu số lần truy cập vào cơ sở dữ liệu

##### **Spring Data JPA**

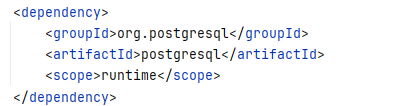
* **Spring JPA (Java Persistence API)** là một phần của Spring Framework và là một phần mở rộng của ORM. Nó cung cấp một cách dễ dàng để làm việc với cơ sở dữ liệu trong ứng dụng Spring. Spring JPA tự động tạo các truy vấn SQL dựa trên các phương thức trong các repository và cung cấp tích hợp mạnh mẽ với Hibernate - một trong các công cụ ORM phổ biến nhất trong cộng đồng Java.
* **Lợi ích JPA:**
* **Giảm đoạn mã và boilerplate**: Spring JPA giúp giảm đoạn mã lặp đi lặp lại và boilerplate code khi làm việc với cơ sở dữ liệu. Các phương thức repository được tạo ra tự động dựa trên tên của phương thức và kiểu dữ liệu của đối tượng.
* **Quản lý thao tác cơ sở dữ liệu**: Spring JPA cung cấp một cách dễ dàng để thực hiện các thao tác cơ sở dữ liệu như tìm kiếm, thêm, sửa và xóa.
* **Tích hợp với Spring Framework**: Spring JPA được tích hợp chặt chẽ với Spring Framework, cho phép sử dụng các tính năng khác của Spring như Dependency Injection, Transaction Management và Security.
* **Cách sử dụng** Spring JPA
* Thêm Spring Boot Starter Data JPA: Thêm Spring Boot Starter Data JPA vào file pom.xml.
* Tạo Entity Classes: Tạo các lớp đối tượng (entity classes) để đại diện cho các bảng trong cơ sở dữ liệu. Sử dụng các chú thích của JPA để xác định mối quan hệ giữa các đối tượng.
* Tạo Repository Interfaces: Tạo các interface repository để thực hiện các thao tác cơ sở dữ liệu. Spring JPA sẽ tự động tạo các truy vấn SQL dựa trên tên phương thức.
* Chạy ứng dụng: Khởi chạy ứng dụng Spring Boot của bạn. Spring JPA sẽ tự động cấu hình và quản lý các repository.

#### **Cấu hình kết nối cơ sở dữ liệu (H2, MySQL, PostgreSQL)**

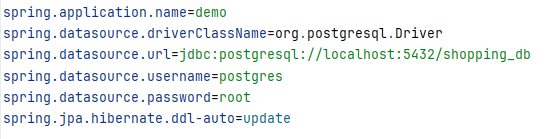
##### **Thêm dependency vào file pom.xml**



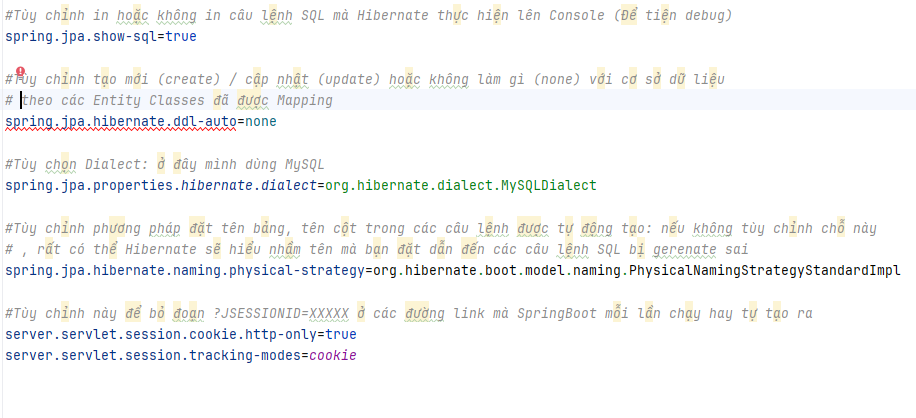
*Một module nhằm đơn giản hóa việc implement JPA trong project*

**

*Thư viện kết nối với cơ sở dữ liệu PostgreSQL*

**

Tạo một cơ sở dữ liệu và cấu hình connection string tại **application.properties**



*Điều chỉnh một số settings khác trong file* ***application.properties***

##### **Tự động tạo các Entities ứng với mỗi bảng trong database**

#### **Khái niệm Entity, Repository**

##### **Entity**

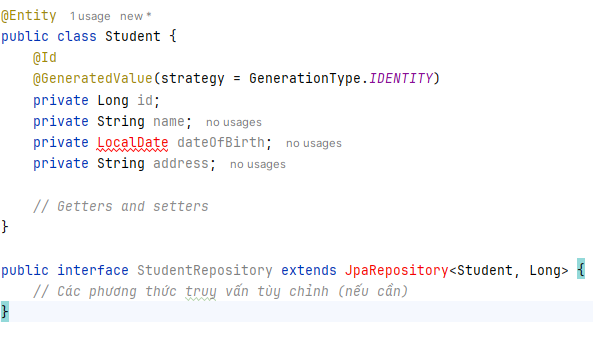
* **Entity** là các đối tượng thể hiện tương ứng 1 table trong cơ sở dữ liệu. Entity thường là các class POJO đơn giản, gồm các phương thức getter, setter.
* **Đặc điểm** của một **Entity**:
* Có thể tương tác với cơ sở dữ liệu quan hệ.
* Được xác định thông qua một định danh (id), tương đương với khóa chính trong table của cơ sở dữ liệu quan hệ.
* Hỗ trợ transaction.
* Hỗ trợ kế thừa giống như những class Java khác.

##### **Repository**

* Tầng Repository, chịu trách nhiệm giao tiếp với cơ sở dữ liệu hoặc các nguồn dữ liệu khác, thực hiện các thao tác truy vấn và cung cấp dữ liệu cho tầng Service.
* Repository giúp tách biệt logic truy cập dữ liệu từ logic nghiệp vụ của ứng dụng, tạo ra một lớp trung gian giữa ứng dụng và nguồn dữ liệu.

1. Sự kết hợp giữa Entity và Repository

* Trong một ứng dụng, các Entity thường được quản lý và thao tác thông qua các Repository. Điều này giúp giữ cho mã nguồn sạch sẽ và dễ bảo trì hơn bằng cách tách biệt rõ ràng giữa logic nghiệp vụ và logic truy cập dữ liệu.



#### **Tạo các thao tác CRUD cơ bản (Create, Read, Update, Delete)**

##### **Create**

##### **Read**

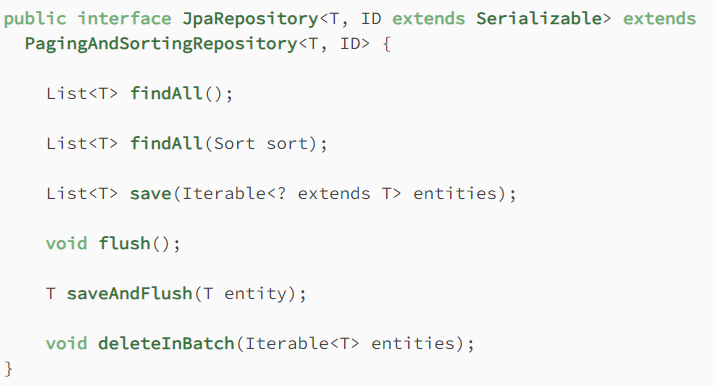
##### **Update**

##### **Delete**

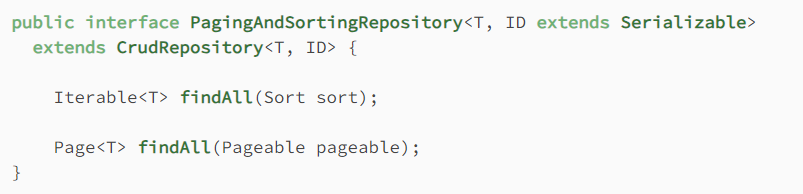
#### **Sử dụng JpaRepository, CrudRepository**

##### **JpaRepository**

* Cung cấp các phương pháp liên quan đến JPA như xóa ngữ cảnh lưu trữ và xóa bản ghi theo đợt.
* JpaRepository chứa toàn bộ API của CrudRepository và PagingAndSortingRepository .



* **findAll()** – lấy danh sách tất cả các thực thể có sẵn trong cơ sở dữ liệu
* **findAll(…)** – lấy danh sách tất cả các thực thể có sẵn và sắp xếp chúng bằng điều kiện được cung cấp
* **save(…)** – lưu một Iterable của các thực thể. Ở đây, chúng ta có thể truyền nhiều đối tượng để lưu chúng trong một đợt
* **flush()** – flush tất cả các tác vụ đang chờ xử lý vào cơ sở dữ liệu
* **saveAndFlush(…)** – lưu thực thể và xóa các thay đổi ngay lập tức
* **deleteInBatch(…)** – xóa một Iterable của các thực thể, có thể truyền nhiều đối tượng để xóa chúng trong một đợt
* Giao diện trên mở rộng **PagingAndSortingRepository**, nghĩa là nó cũng có tất cả các phương thức có trong **CrudRepository** .



Giao diện này cung cấp phương thức findAll(Pageable pageable) , đây là chìa khóa để triển khai Phân trang.

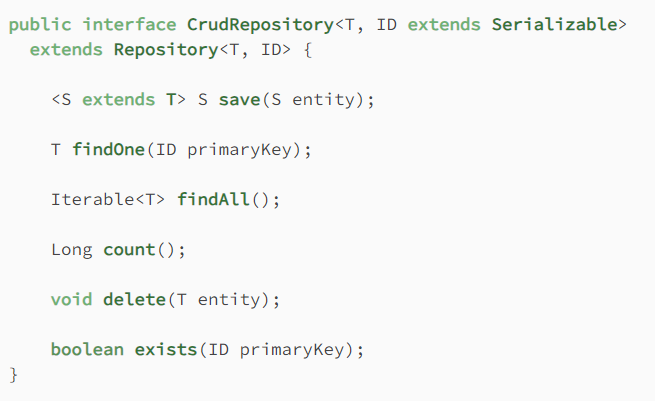
* Khi sử dụng **Pageable** , chúng ta tạo một đối tượng ***Pageable*** với một số thuộc tính nhất định và chúng ta phải chỉ định ít nhất những điều sau:
* Kích thước trang
* Số trang hiện tại
* Phân loại
* Vì vậy, hãy giả sử rằng chúng ta muốn hiển thị trang đầu tiên của một tập kết quả được sắp xếp theo *họ,* tăng dần, mỗi trang không quá năm bản ghi. Đây là cách chúng ta có thể thực hiện điều này bằng cách sử dụng **PageRequest** và định nghĩa *Sort* :



* Truyền đối tượng có thể phân trang vào truy vấn dữ liệu Spring sẽ trả về kết quả cần tìm (tham số đầu tiên của *PageRequest* bắt đầu từ số không).

##### **CrudRepository**

* Khi không cần đến đầy đủ chức năng được cung cấp bởi **JpaRepository** và **PagingAndSortingRepository** , có thể sử dụng **CrudRepository** .



* **Chức năng CRUD**:
* **save(…)** – lưu một Iterable của các thực thể. Ở đây, chúng ta   
  có thể truyền nhiều đối tượng để lưu chúng trong một đợt
* **findOne(…)** – lấy một thực thể duy nhất dựa trên giá trị khóa chính đã truyền
* **findAll()** – lấy Iterable của tất cả các thực thể có sẵn trong cơ sở dữ liệu
* **count()** – trả về số lượng tổng số thực thể trong một bảng
* **delete(…)** – xóa một thực thể dựa trên đối tượng đã truyền
* **exists(…)** – xác minh xem một thực thể có tồn tại hay không dựa trên giá trị khóa chính đã truyền
* Cung cấp tất cả các truy vấn trừu tượng cơ bản cần thiết trong một ứng dụng.

### **Ngày 4: RESTful API**

#### **Khái niệm RESTful API**

##### **Tổng quan**

* **RESTful API (Representational State Transfer API)** là một kiểu kiến trúc cho các API (Application Programming Interface) được sử dụng để truyền tải và trao đổi dữ liệu giữa các ứng dụng web. RESTful API sử dụng giao thức HTTP để truyền tải dữ liệu giữa máy chủ và máy khách, và sử dụng các phương thức HTTP như GET, POST, PUT và DELETE để thực hiện các thao tác trên tài nguyên.

##### **Lợi ích**

* Sử dụng các URL dễ đọc và dễ hiểu, và sử dụng các định dạng dữ liệu như JSON hoặc XML để trao đổi thông tin giữa máy chủ và máy khách.
* **RESTful API** cũng có tính khả di động cao, cho phép các ứng dụng khác nhau có thể truy cập và sử dụng các tài nguyên một cách dễ dàng.
* **Đơn giản hóa:**
* Kết nối và trao đổi giữa các ứng dụng đơn giản.
* Lập trình viên chỉ cần tập trung vào các chức năng chính của ứng dụng thay vì quản lý các thành phần khác nhau.
* **Tính mở rộng:**
* Các ứng dụng có thể kết nối và trao đổi dữ liệu dễ dàng giúp cho việc mở rộng ứng dụng trở nên thuận tiện hơn.
* **Tính độc lập:**
* Các ứng dụng hoạt động độc lập với nhau giúp cho việc bảo trì, nâng cấp trở nên thuận tiện hơn.
* **Khả năng tương thích:**
* RESTful API có thể tương tích với các ứng dụng khác nhau, việc tích hợp các ứng dụng với nhau trở nên dễ dàng, khi có thể sử dụng cùng một API để tương tác với các dữ liệu khác nhau mà không cần tạo ra các API riêng biệt.

##### **Nguyên tắc**

* Dựa trên kiểu tương tác **Client-Server**, các tài nguyên (**resources**) của **server** được truy cập và sử dụng bởi **Client** thông qua các **giao thức** **HTTP**.
* **Resources** là một đối tượng được quản lý bởi server, mỗi tài nguyên được đại diện bởi **URI (Uniform Resource Identifier)**.
* **URI** được sử dụng để định danh các tài nguyên trên server, cũng như yêu cầu để thực hiện các thao tác trên tài nguyên đó.
* **Representation** đại diện bằng một định dạng cụ thể **(HTML, JSON, XML, YAML)**, định dạng này dùng để truyền thông tin giữa client và server.
* **Metadata** để cung cấp thông tin về các tài nguyên như **định dạng, ngôn ngữ, thời gian tạo và sửa đổi** của tài nguyên, sử dụng để **quản lý** các **tài nguyên** trên server.

##### **Các bước triển khai**

* **Xác định các tài nguyên** (resources) của API.
* **Thiết kế các URL** để xác định tài nguyên truy cập.
* **Sử dụng các phương thức HTTP** để thực hiện các thao tác trên tài nguyên.
* **Quyết định dạng dữ liệu API** sẽ sử dụng (JSON, XML..) để trao đổi dữ liệu.
* **Quản lý lỗi** với các mã lỗi và thông điệp rõ ràng để client có thể xử lý.
* **Bảo mật API** để đảm bảo chỉ các user được ủy quyền mới có thể truy cập tài nguyên với cơ chế xác thực và phân quyền.
* **Tài liệu hóa các API** để giúp người dùng hiểu rõ cách sử dụng API.

#### **Tạo các endpoint RESTful với Spring Boot**

* Khi app gọi đến URL https://abc.com/au/1 (/au/1 là 1 endpoint).
* **Endpoint** là một phần quan trọng trong quá trình phát triển API.

##### **Tạo các endpoint với HTTP GET:**

* GET /resources – Tìm một danh sách records từ resource có thể có phân trang thì tốt nhất hoặc lấy toàn bộ records.
* GET /resources/X – Chỉ cần lấy Record X, ví dụ /users/1 .
* GET /resources/X,Y,Z – Người dùng muốn tìm kiếm trên nhiều điều kiện hay 1 điều kiện mà nhiều giá trị.
* GET /places/X/users – Lấy tất cả người dùng đang sinh sống trong vùng X.
* GET /users/X/places – Lấy tất cả các nơi mà người dùng này đang hay đã ở.
* GET /users/X/places/Y – Tìm kiếm user theo điều kiện X và Places theo điều kiện Y.

##### **Tạo các endpoint với HTTP DELETE:**

* DELETE /users/X – Xóa một người dùng.
* DELETE /users/X,Y,Z – Xóa một vài dòng dữ liệu.
* DELETE /users – Xóa tất cả các dòng dữ liệu (**Không nên**).
* DELETE /users/X/image – Xóa image của người dùng X.
* DELETE /users/X/images – Xóa một loạt các hình ảnh của người dùng X.

##### **Tạo các endpoint với HTTP POST và PUT:**

* PUT /resources.
* POST/resources.
* Lưu ý:
* Khi thiết kế các **endpoint** chỉ nên chứa resources (danh từ), không phải hành động (action) hoặc động từ bởi các giao thức HTTP đã đóng vai trò như động từ.
* **Resource** phải luôn ở dạng số nhiều trong API endpoint và nếu muốn truy cập một instance của resource.
* **POST** dùng tạo 1 dòng dữ liệu mới còn **PUT** thì dùng cập nhật một dòng dữ liệu đã có.

#### **Các phương thức HTTP (GET, POST, PUT, DELETE)**

* **HTTP Request** là tập hợp thông tin được gửi từ các máy khách (client) đến máy chủ (server). Nó là những yêu cầu cần máy chủ tìm kiếm hoặc xử lý và phản hồi kết quả lại client.
* Các yêu cầu HTTP được gửi đến có thể là các file dưới dạng **XML hoặc Json**. Và đó cũng là những định dạng mà cả hai phía **client – server** đều có thể hiểu được.
* Bất cứ **HTTP Request** nào cũng có cấu trúc cụ thể. Nó gồm **3 thành phần** chính. Đó là **Request line, Request header và Body Request**.

##### **Phương thức GET**

* **Get** là phương thức được máy khách sử dụng để gửi dữ liệu tới máy chủ thông qua URL. Phương thức này gửi trên thanh địa chỉ của Browser.
* Máy chủ sẽ nhận yêu cầu từ đường dẫn này và trả kết quả ngược trở lại máy khách client. Phương thức này được sử dụng rất phổ biến và nó không cần phải có Request body.
* Đặc điểm:
* Độ dài các giá trị được giới hạn trong 255 kí tự.
* Chỉ các dữ liệu kiểu String mới được hỗ trợ.
* Thông tin dữ liệu yêu cầu được lưu vào bộ nhớ cache.
* Lịch sử trình duyệt lưu trữ các tham số được truyền vào.
* Thông tin được đánh dấu và xem lại dễ dàng trên lịch sử trình duyệt.

##### **Phương thức POST**

* Phương thức **Post** được sử dụng để gửi dữ liệu đến server. Người dùng có thể thêm dữ liệu mới hoặc cập nhật dữ liệu vào database.
* Đặc điểm:
* Các dữ liệu bổ sung sẽ không xuất hiện trong URL của trình duyệt.
* Dữ liệu không được lưu trong lịch sử của trình duyệt.
* Post không giới hạn độ dài ký tự và hỗ trợ nhiều kiểu dữ liệu khác nhau (String, integers, binary).

##### **Phương thức PUT**

* **Put** là phương thức có chức năng rất giống với Post. Tuy nhiên chủ yếu nhằm mục đích cập nhật các dữ liệu đã có trong database. Người dùng bắt buộc phải cập nhật dữ liệu mới thay thế dữ liệu cũ của đối tượng.

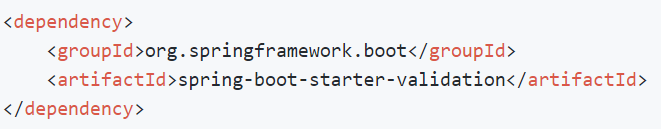
##### **Phương thức DELETE**

* **Delete** là phương thức dùng để xóa các dữ liệu tìm kiếm từ server về tài nguyên thông qua URL. Phương thức này cũng được sử dụng nhiều bởi thời gian xử lý nhanh do không chứa Request body.

#### **Validation với @Valid, @NotNull, @Size.**

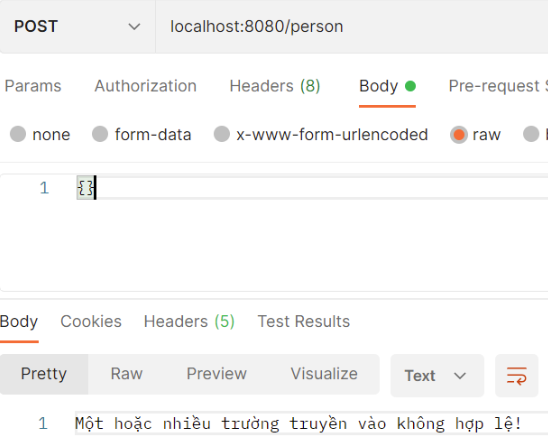
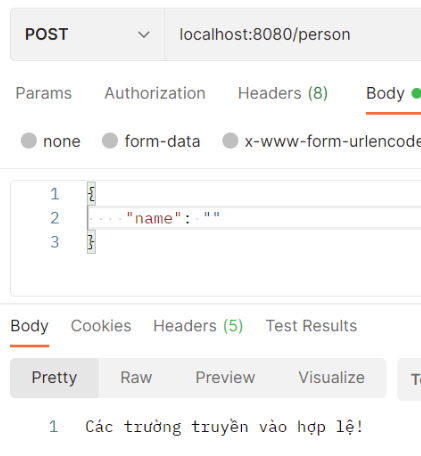
##### **Khai báo dependency**

* Sử dụng thư viện **spring-boot-starter-validation** để thực hiện xác thực.

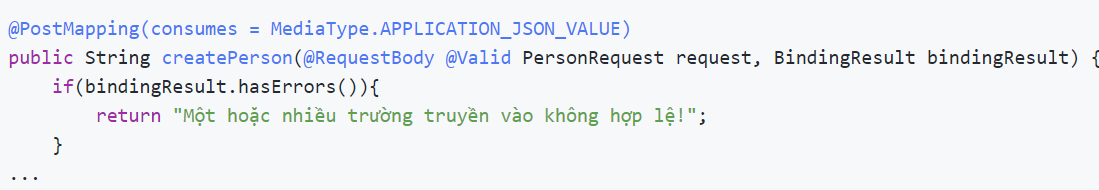
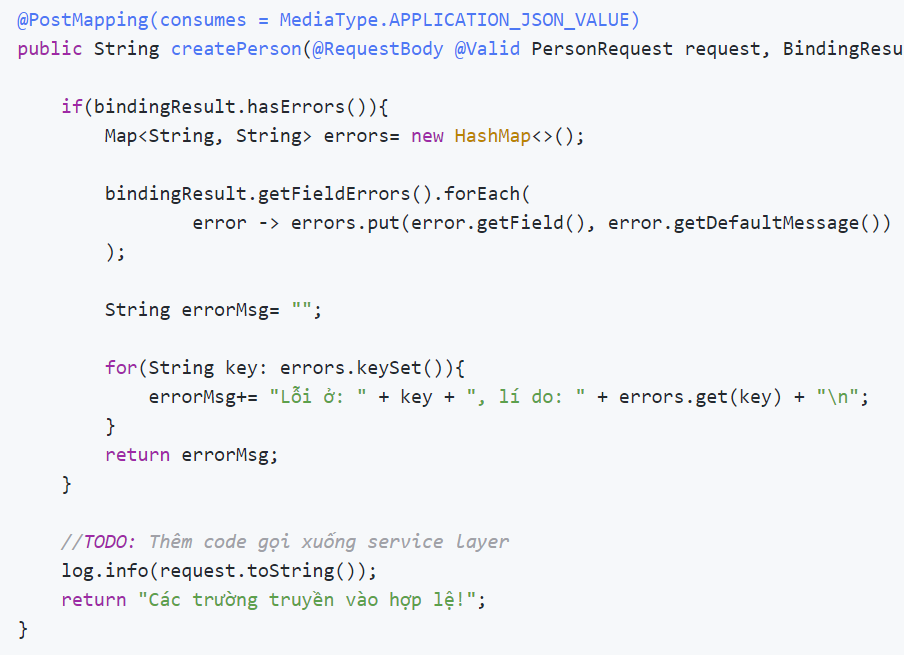
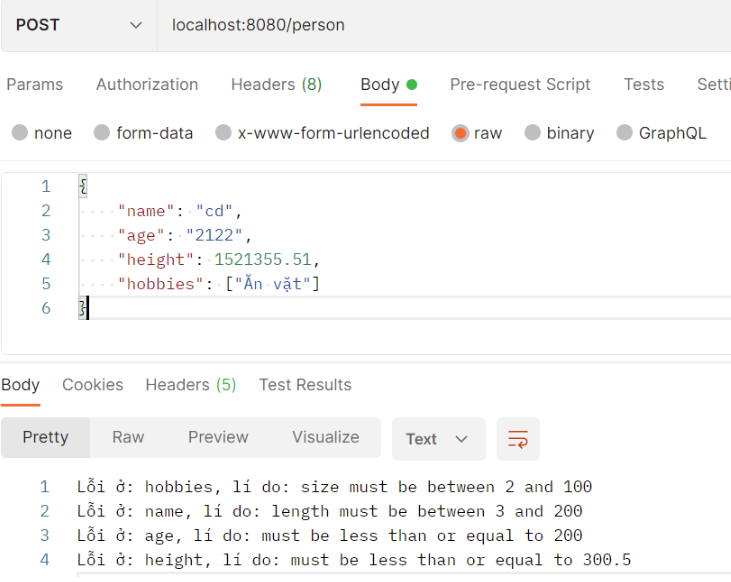


* Chú ý:
* Một số nguồn nói rằng spring-boot-starter-validation đã có sẵn trong dependency spring-boot-starter-web. Tuy nhiên điều này không còn đúng từ Spring Boot 2.3 và trở đi: lập trình viên sẽ phải tự khai báo thư viện validation của Spring như trên.

##### **@Valid, @NotNull, @Size**

* Spring Validation cung cấp các annotation (chú thích) trong package javax.validation.constraints để hỗ trợ việc xác thực dữ liệu.
* Nếu như trong request truyền vào không có trường sử dụng **@NotNull**
* Tuy nhiên nếu như tên được truyền vào là một String rỗng ("")
* Annotation **@NotNull** kiểm tra xem liệu thuộc tính có null hay không. Nhưng không kiểm tra được rỗng vì thế còn có thêm **@NotBlank** và **@NotEmpty**:
* **@NotNull**: có thể dùng được cho hầu như tất cả các kiểu dữ liệu trong Java, từ chối giá trị null hoặc không có nhưng chấp nhận giá trị rỗng. Note: Thậm chí annotation này có thể dùng được cho các kiểu dữ liệu nguyên thủy (int, long, float, double, char, byte...) mà không báo lỗi (mặc dù các kiểu nguyên thủy không thể null) nhưng mà nó sẽ không có tác dụng gì cả.
* **@NotEmpty**: có thể dùng được cho CharSequence (interface của String, StringBuffer và StringBuilder), Collection, Map, Array. Annotation này từ chối cả giá trị null lẫn giá trị rỗng bằng cách kiểm tra độ dài (length) (String) hoặc kích thước (size) (Collection) xem có lớn hơn 0 hay không.
* **@NotBlank**: có thể dùng được cho String. Annotation này từ chối String có giá trị null và String có độ dài là 0 sau khi đã trim (loại bỏ hết khoảng trắng thừa ở đầu và cuối của String).

##### **Xử lý lỗi xác thực bằng BindingResult**

* **bindingResult** là nơi Spring chứa kết quả của việc xác thực dữ liệu. Với ví dụ của chúng ta, nếu như có lỗi trong quá trình xác thực đầu vào thì chúng ta có thể kiểm tra bằng **BindingResult.hasErrors()**.
* Để xử lý tinh tế hơn thì phải sử dụng **BindingResult.getFieldErrors** để lấy về tất cả các trường bị lỗi xác thực
* Để tinh chỉnh tin nhắn lỗi trả ra theo ý muốn, bổ sung giá trị message cho các annotation

### **Ngày 5: Spring Security**

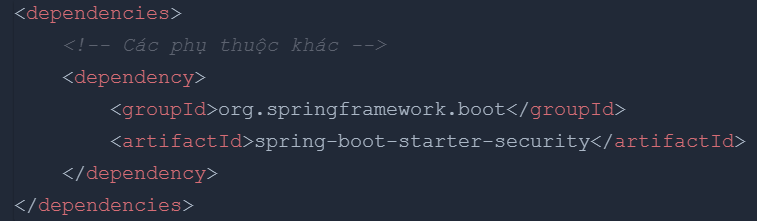
#### **Giới thiệu về Spring Security**

* Một trong những tính năng cốt lõi của Spring Framework, nó cho phép bạn quản lý việc phân quyền và xác thực người dùng trước khi cho họ truy cập vào các tài nguyên của ứng dụng web.

#### **Cấu hình bảo mật cơ bản**

##### **Cài đặt**

* Cần cài đặt các phụ thuộc cần thiết cho dự án Spring Boot.



##### **Triển khai cơ bản**

* Để bắt đầu sử dụng Spring Security, cần kích hoạt tính năng này trong ứng dụng Spring Boot, có thể làm điều này bằng cách thêm annotation **@EnableWebSecurity** lên một bean bất kỳ trong ứng dụng.



* Tạo một tài khoản người dùng có tên đăng nhập "user" và mật khẩu "password" bằng cách sử dụng phương thức **User.withDefaultPasswordEncoder().**
* Sử dụng **.authorizeRequests()** để định cấu hình quyền truy cập.
* **.antMatchers()** cho phép chúng ta xác định các URL mà người dùng có thể truy cập mà không cần xác thực.
* **.anyRequest().authenticated()** yêu cầu xác thực đối với tất cả các URL còn lại.
* **.formLogin()** cho phép xác thực người dùng bằng cách sử dụng một trang đăng nhập tùy chỉnh.
* **.loginPage("/login")** xác định trang đăng nhập tùy chỉnh. Nếu bạn không cung cấp trang đăng nhập, **Spring Security** sẽ tạo một trang đăng nhập mặc định.
* **.logout()** cho phép người dùng đăng xuất.

#### **Xác thực người dùng với in-memory Authentication và JDBC Authentication**

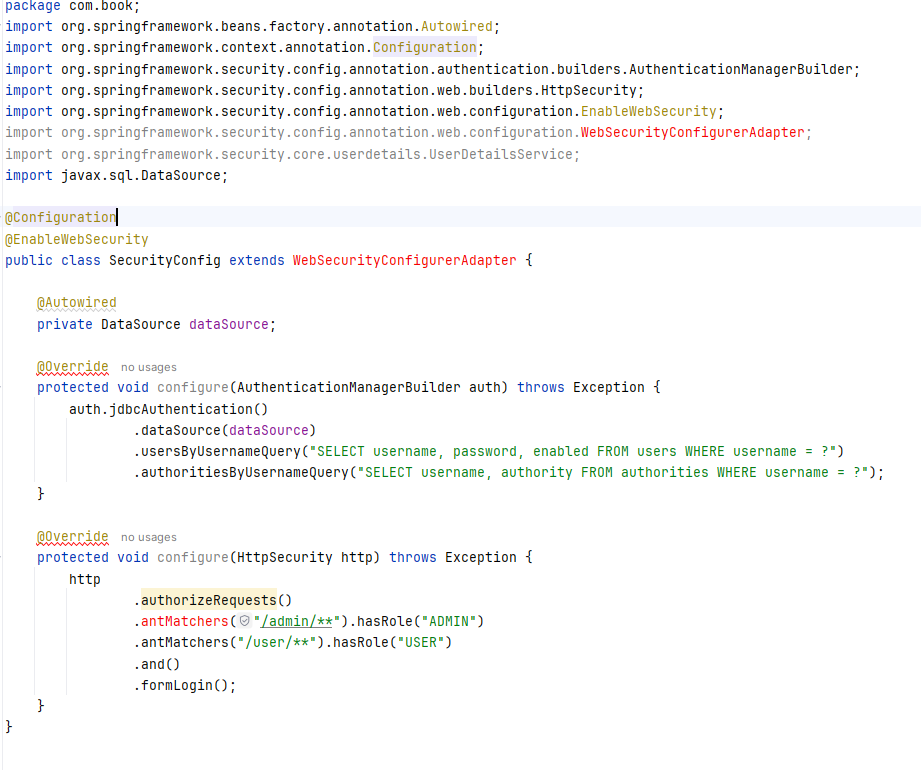
##### **In-memory Authentication**

* Phương pháp xác thực người dùng được cấu hình ngay trong mã nguồn, dữ liệu người dùng được lưu trữ tạm thời trong bộ nhớ (RAM). Đây là phương pháp đơn giản và nhanh chóng để thử nghiệm và phát triển ứng dụng.
* Cấu hình In-memory Authentication:



##### **JDBC Authentication**

* Sử dụng cơ sở dữ liệu để lưu trữ thông tin người dùng. Phương pháp này thường được sử dụng trong các ứng dụng thực tế vì nó dễ dàng quản lý và mở rộng.
* Cấu hình JDBC Authentication:



#### **Giới thiệu về JWT**

* một chuẩn mở (RFC 7519) định nghĩa cách thức truyền tải thông tin an toàn dưới dạng đối tượng JSON giữa các bên. JWT thường được sử dụng để xác thực và ủy quyền trong các ứng dụng web.
* Một JWT bao gồm ba phần chính:
* **Header**: Chứa thông tin về loại token và thuật toán ký.
* **Payload**: Chứa dữ liệu người dùng và các claims.
* **Signature**: Được tạo ra từ header, payload và một khóa bí mật.

Phần **Header** sẽ chứa kiểu dữ liệu , và thuật toán sử dụng để mã hóa ra chuỗi JWT

{

"typ": "JWT",

"alg": "HS256"

}

* “typ” (type) chỉ ra rằng đối tượng là một JWT
* “alg” (algorithm) xác định thuật toán mã hóa cho chuỗi là HS256

**Payload** sẽ chứa các thông tin mình muốn đặt trong chuỗi Token như username , userId , author , … ví dụ:

{

"user\_name": "admin",

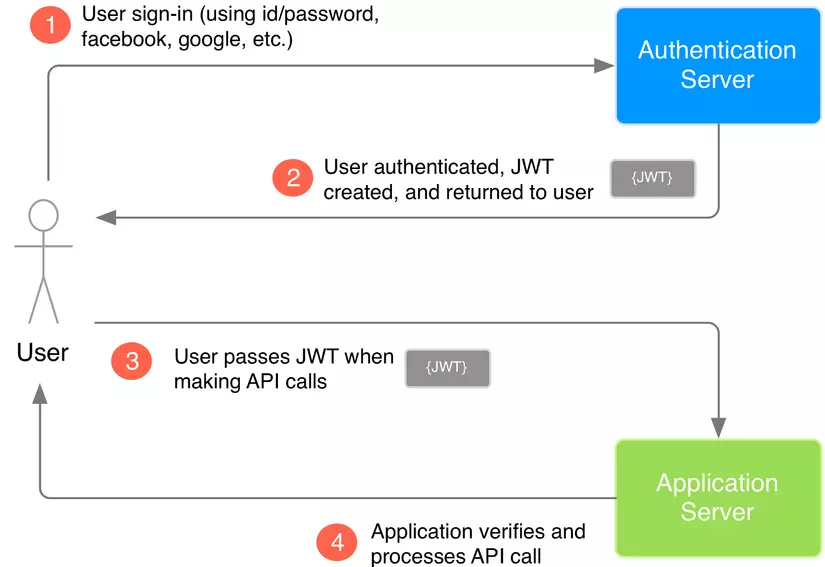
"user\_id": "123",

"authorities": "ADMIN\_USER",

"jti": "474cb37f-2c9c-44e4-8f5c-1ea5e4cc4d18"

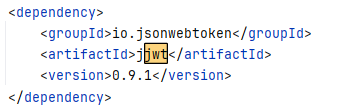
}

**Signature**: Phần chữ ký này sẽ được tạo ra bằng cách mã hóa phần header, payload kèm theo một chuỗi secret (khóa bí mật) , ví dụ: signature = Hash( data, secret );

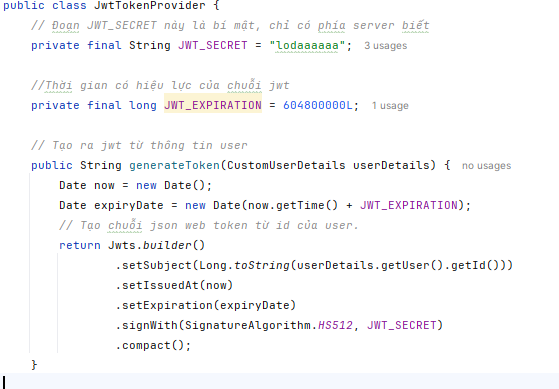
* Lưu ý:
* Không đặt quá nhiều thông tin trong chuỗi **payload** vì nó sẽ ảnh hưởng đến độ trễ khi Server phải xác nhận một Token quá dài.
* Khi nào nên sử dụng JWT:
* **Authentication:** Đây là kịch bản phổ biến nhất cho việc sử dụng JWT.
* **Trao đổi thông tin:** JSON Web Token là 1 cách thức không tồi để truyền tin an toàn giữa các thành viên với nhau, nhờ vào phần "chữ ký" của nó.
* Cách hoạt động của JWT:
* Trong quá trình xác thực, người dùng đăng nhập thành công bằng cách sử dụng các thông tin của họ (email or username, password), **JSON Web Token** sẽ được trả lại và phải được lưu lại dưới local (thường là trong local storage, nhưng có lúc cookie cũng có thể được sử dụng).
* Bất cứ khi nào người dùng muốn truy cập vào router hoặc tài nguyên cần có quyền, họ phải gửi JWT trong Authorization header sử dụng Bearer schema như sau: **Authorization: Bearer <token>**

#### **Tạo và xác thực JWT Token**

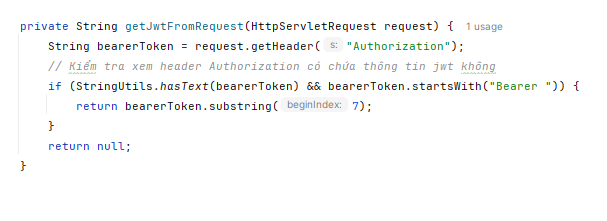
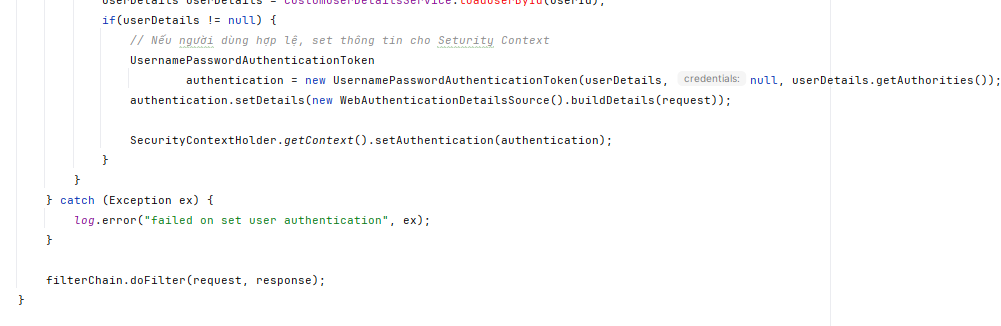
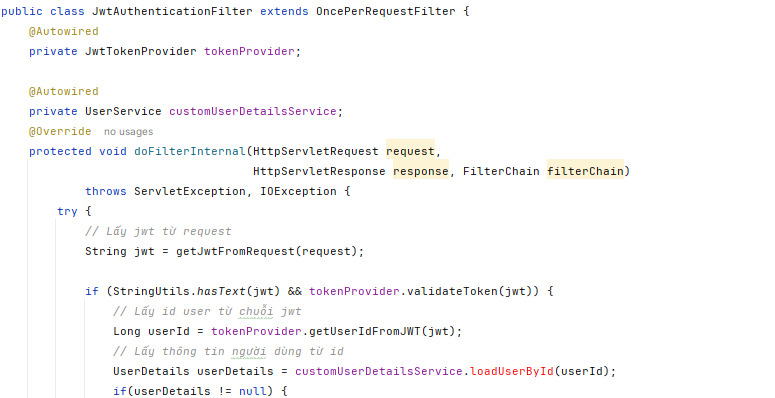
##### **Cài đặt pom.xml**



##### **Tạo JWT Token**



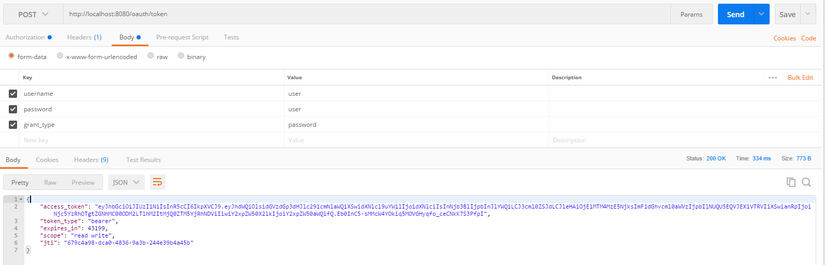
##### **Xác thực JWT Token**

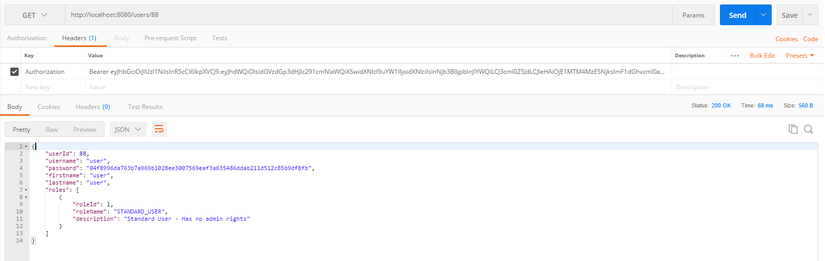


#### **Bảo vệ các endpoint bằng JWT**

* Bước 1 : Người dùng yêu cầu đăng nhập với Username , Password
* Bước 2 + 3 : Server nhận được yêu cầu và kiểm tra Username , Password nếu đúng sẽ gửi cho người dùng một chuỗi JWT
* Bước 4 : Người dùng sẽ dùng JWT này kèm theo các yêu cầu kế tiếp
* Bước 5 + 6 : Server sẽ nhận yêu cầu và kiểm tra chuỗi JWT , nếu chuỗi hợp lệ thì sẽ thực hiện yêu cầu.

1. **Login**

* Khi đăng nhập thành công thì Server sẽ trả về một chuỗi access\_token.

1. **Request**

* Sau khi nhận được Token rồi, sẽ gửi yêu cầu Server lấy thông tin của User này kèm theo Token được đóng gói ở Header.