BÁO CÁO

Contents

A/ Java

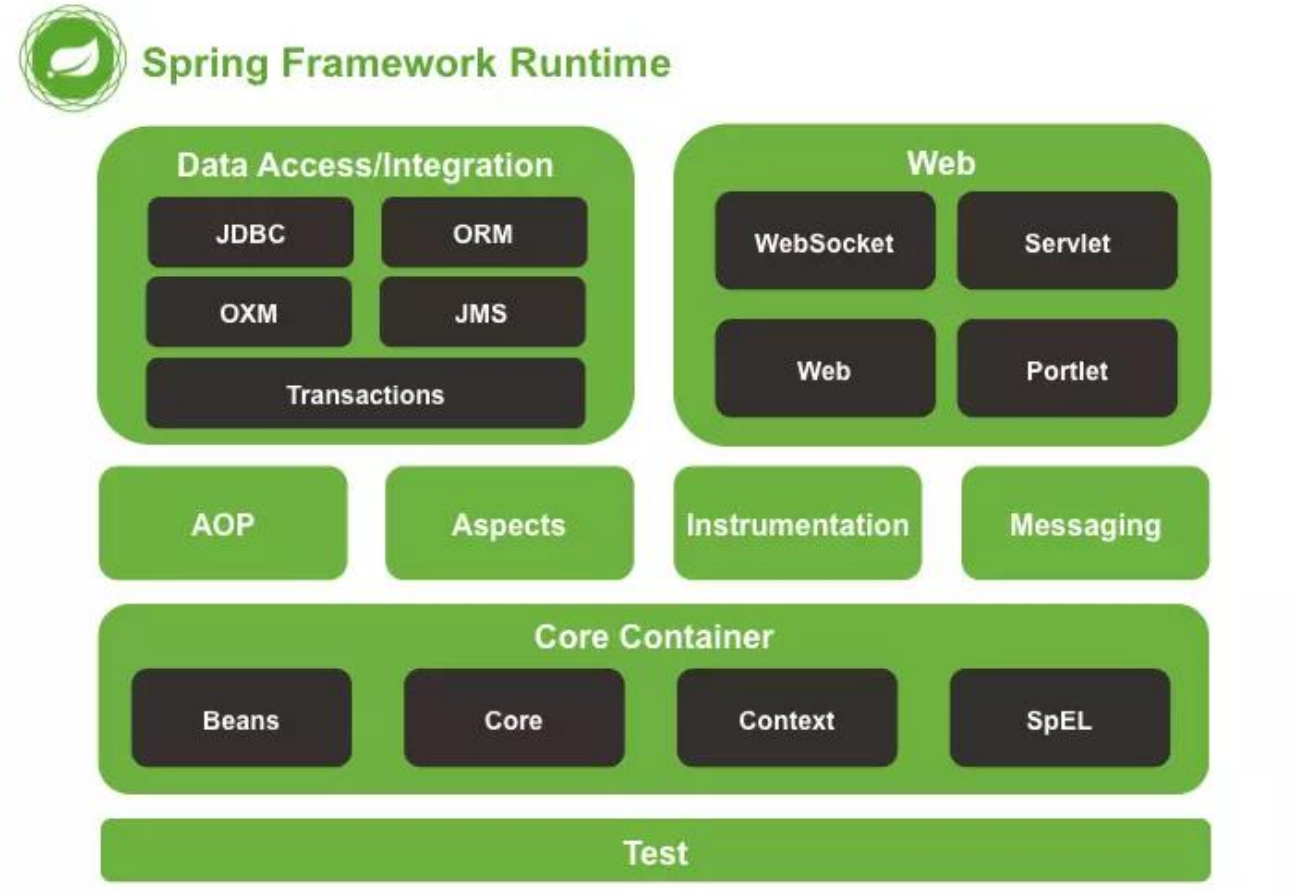
I. Framework

Vấn đề: một project có quá nhiều dependency => inject vất vả => Xuất hiện framework (kho chứa các dependency) sẽ tự động inject (@Autowired). Điển hình là Spring framework: mục đích tạo ra container chứa dependency => Rút ngắn thời gian lập trình và test, giảm sự rườm rà

Framework là một bộ công cụ và thư viện phát triển được thiết kế để giúp việc xây dựng ứng dụng trở nên dễ dàng hơn. Nó cung cấp một cấu trúc, các tiêu chuẩn và các công cụ giúp lập trình viên tập trung vào việc triển khai business logic thay vì phải viết lại các thành phần cốt lõi của ứng dụng.

1. Spring framework

Spring Framework là một trong những framework phát triển ứng dụng Java phổ biến nhất. Nó cung cấp một cách tiếp cận mô hình phát triển linh hoạt và mạnh mẽ, giúp xây dựng các ứng dụng doanh nghiệp và ứng dụng web hiệu quả.



2. Springboot

Spring Boot là một extension của Spring Framework, giúp loại bỏ các bước cấu hình phức tạp mà Spring bắt buộc. Spring Boot là dự án phát triển bởi Java (JAV) trong hệ sinh thái Spring framework.

Một số tính năng nổi bật của Spring Boot đó là:

- Tạo ra các ứng dụng Spring mang tính độc lập.
- Nhúng trực tiếp Tomcat, Jetty hoặc Undertow mà không cần phải triển khai ra file WAR.
- Starter dependency giúp cho việc chạy cấu hình Maven trở nên đơn giản hơn.
- Tự động chạy cấu hình Spring khi cần thiết.
- Không sinh code cấu hình, đồng thời không yêu cầu phải cấu hình bằng XML.

Spring có kích thước rất nhỏ, trong suốt và nhẹ trong quá trình chạy. Vì vậy nên, đây là một giải pháp khá gọn, nhẹ với khả năng hỗ trợ để tạo ra và phát triển các ứng dụng web có mã hiệu suất cao. Hơn hết, có thể dễ dàng kiểm tra, thử nghiệm hoặc tái sử dụng code.

Hai nguyên tắc thiết kế chính để xây dựng nên spring framework đó là:

- Dependency Injection (DI)
- Aspect Oriented Programming (AOP)

Những tính năng cốt lõi của Spring framework có thể được sử dụng trong việc phát triển java desktop, java web, … Mục tiêu chính là dễ dàng phát triển các ứng dụng J2EE dựa trên mô hình sử dụng POJO

❖ IoC

IoC (Inversion of Control) là một nguyên tắc trong lập trình, tuân thủ theo nguyên tắc D (SOLID), trong đó quyền kiểm soát các đối tượng và luồng của chương trình được chuyển từ phần mềm chính tới một framework hoặc container. Thay vì phần mềm gọi các hàm hoặc khởi tạo các đối tượng, hàng rào của framework sẽ gọi các hàm và khởi tạo các đối tượng để tiếp quản và điều khiển luồng của chương trình => Tăng tính linh hoạt, dễ bảo trì và tái sử dụng mã nguồn. Có thể triển khai IoC thông qua các cơ chế như Strategy design pattern, Service Locator pattern, Factory pattern, và Dependency Injection (DI).

IoC giúp tách biệt và quản lý phụ thuộc giữa các thành phần trong ứng dụng, làm cho mã của bạn linh hoạt hơn và dễ dàng thay đổi hoặc kiểm thử.

❖ DI

Vậy dependency là gì? dependency (phụ thuộc) đề cập đến một thành phần hoặc một đối tượng mà một thành phần khác cần để hoạt động. Đối tượng hoặc thành phần này có thể là một đối tượng, một interface, một thư viện, hoặc bất cứ thứ gì mà thành phần khác cần sử dụng để đạt được mục tiêu của nó.

Thành phần phụ thuộc luôn được sử dụng hoặc được truyền vào thành phần khác để thực hiện chức năng cần thiết

DI (Dependency Injection) là một cách triển khai của IoC (giống IoC). Nó tập trung vào việc cung cấp các phụ thuộc (dependencies) cho một đối tượng từ bên ngoài. Thay vì đối tượng tự tạo hoặc tìm kiếm những đối tượng phụ thuộc của nó. Các phụ thuộc này có thể là các đối tượng, các giao diện, hoặc các giá trị cấu hình => giúp tách biệt các thành phần và làm cho chúng dễ dàng thay thế và kiểm soát.

Không nên dùng như này:

```
public class UserRepository {
   private DatabaseConnection dbConnection;

   public UserRepository() {
      this.dbConnection = new DatabaseConnection();
   }
}
```

UserRepository trực tiếp tạo ra và nắm quyền điều khiển DatabaseConnection. Khi muốn thay đổi sang MySQLDatabaseConnection hoặc PostgreSQLDatabaseConnection thì phải thay đổi trong UserRepository => bất tiện

Nên dùng như này:

```
public class UserRepository {
   private DatabaseConnection dbConnection;

   public UserRepository(DatabaseConnection dbConnection) {
      this.dbConnection = dbConnection;
   }
}
```

UserRepository không tạo trực tiếp một đối tượng DatabaseConnection, nhưng nó được cung cấp từ bên ngoài qua constructor của UserRepository. Điều này làm linh hoạt trong việc thay đổi các giao diện và triển khai của DatabaseConnection mà không ảnh hưởng đến UserRepository.

❖ Bean

Spring boot sử dụng khái niệm "bean" để đại diện cho các đối tượng và được quản lý bởi Spring Container.

Một số loại bean:

- Singleton Bean: Là loại bean mặc định trong Spring. Một singleton bean chỉ có một phiên bản duy nhất được tạo ra và được chia sẻ cho toàn bộ ứng dụng.
- Prototype Bean: Đối với mỗi yêu cầu, một bean prototype sẽ tạo ra một phiên bản mới. Mỗi bean được tạo ra sẽ thành một đối tượng hoàn toàn độc lập.
- Request Scope Bean: Bean với request scope chỉ tồn tại trong suốt quá trình xử lý một yêu cầu HTTP cụ thể.
- Session Scope Bean: Bean với session scope chỉ tồn tại trong suốt chuỗi thời gian kết nối của một phiên (session) HTTP.

- Global Session Scope Bean: Bean với global session scope chỉ tồn tại trong suốt chuỗi thời gian kết nối của một phiên (session) toàn cầu.
- Application Scope Bean: Bean với application scope tồn tại trong toàn bộ vòng đời của ứng dụng. Chúng được tạo một lần duy nhất khi ứng dụng được khởi động.

❖ Container

Trong Spring Framework, Spring IoC (Inversion of Control) Container là một phần quan trọng trong Spring để quản lý vòng đời và dependencies của các đối tượng trong ứng dụng. Nó có vai trò quản lý đối tượng, có trách nhiệm tạo ra, cấu hình và quản lý các đối tượng bean của ứng dụng. Thay vì tái tạo đối tượng bằng cách gọi trực tiếp từ mã, ta chỉ cần định nghĩa các đối tượng trong cấu hình (tệp XML hoặc thông qua Java Config) và container sẽ tự động khởi tạo, khởi tạo dependencies (đảm bảo dependencies được đưa vào bean một cách tự động, dễ dàng thiết lập và thay thế), quản lý và giải phóng tài nguyên khi không cần thiết nữa.

Bean Factory và ApplicationContext đều là hai thành phần quan trọng của Spring IoC Container:

- Bean Factory là một phần nhỏ nhất của Spring IoC Container. Nó là interface cung cấp các chức năng cơ bản để tạo và quản lý các bean. Bean Factory chịu trách nhiệm cho việc tải và cấu hình các đối tượng (bean) từ các nguồn dữ liệu như tệp XML hoặc các nguồn dữ liệu khác.
- ApplicationContext (extends từ interface BeanFactory). Nó cung cấp tất cả các chức năng của Bean Factory, nhiều tính năng tiện ích bổ sung và là phiên bản nâng cao hơn. BeanFactory tạo bean khi gọi getBean(), còn ApplicationContext tự động tạo sẵn. Ngoài ra, ApplicationContext hỗ trợ cho việc quản lý giao dịch, xử lý sự kiện, quản lý thông điệp, cấu hình dựa trên annotation và tích hợp tốt với các tính năng của Spring như AOP (Aspect-Oriented Programming) và được sử dụng phổ biến hơn Bean Factory.

  Hai lớp cụ thể của ApplicationContext trong Spring Framework:

  - ClassPathXmlApplicationContext (đọc cấu hình từ file XML trên classpath). Nghĩa là, cần cung cấp các tệp XML cấu hình trong ứng dụng và đặt chúng trong classpath để ClassPathXmlApplicationContext có thể tìm thấy và sử dụng để cấu hình các bean
  - AnnotationConfigApplicationContext (sử dụng cấu hình dựa trên Java annotations, ví dụ như: @Configuration, @ComponentScan, @Bean, …). AnnotationConfigApplicationContext quét các packages đã được chỉ định và tự động tìm kiếm các bean được đánh dấu bởi các annotations và cấu hình chúng

❖ Annotation

Annotation (chú thích) là một tính năng quan trọng, được sử dụng rộng rãi trong lập trình Java, cho phép thêm các thông tin bổ sung vào mã nguồn, giúp trình biên dịch và các công cụ phát triển hiểu và xử lý mã nguồn một cách thông minh.

⇨ Annotation được sử dụng để đánh dấu và cung cấp metadata cho các lớp, phương thức, biến, hoặc gói.

Annotation không ảnh hưởng đến hoạt động chương trình khi chạy, nhưng mang thông tin quan trọng về cấu trúc và mục đích của mã nguồn => lợi cho việc tự động hóa các tác vụ, kiểm tra mã nguồn, và cung cấp hướng dẫn cho các công cụ phát triển.

Cú pháp cơ bản của Annotation: Được đặt trong một dấu @, theo sau là tên của annotation. Một số annotation có thể có giá trị được đặt trong dấu ngoặc đơn (value = ...), nhưng nếu chỉ có một giá trị và không cần gán tên, bạn có thể viết trực tiếp giá trị đó.

Một số annotation phổ biến: @Autowired, @Component, @Override, ...

❖ Cách IoC hoạt động trong Spring: IoC được hiện thực thông qua DI trong spring
- Trong Spring, các beans và quan hệ phụ thuộc của chúng được định nghĩa thông qua các annotation.
- Khi run, Spring tự động phân tích các đối tượng và các dependency của chúng, sau đó đưa vào container, khi cần sẽ tự động inject (@Autowired) các đối tượng thông qua DI. => giảm sự phụ thuộc mạnh mẽ giữa các đối tượng và cho phép dễ dàng thay đổi
- @Configuration là một Annotation đánh dấu trên một Class cho phép Spring Boot biết được đây là nơi định nghĩa ra các Bean.
- Tuỳ chỉnh cấu hình cho Spring Boot chỉ tìm kiếm các bean trong một package nhất định, sử dụng @ComponentScan("<link url>")

❖ Define bean với annotations
- @Component: quản lý bởi Spring và thông báo cho Spring rằng nó cần tạo một instance của class này và quản lý nó
- @Controller: Xác định rằng class này xử lý các request HTTP và trả về một response.
- @RestController: Kết hợp hai annotation @Controller và @ResponseBody. Đánh dấu một class là một controller và method trả về dữ liệu dạng JSON hoặc XML.
- @RequestMapping: Chỉ định URL mà một phương thức trong controller xử lý. Xác định HTTP method (GET, POST, PUT, DELETE...)

- @Autowired: Được sử dụng để tiêm các dependency tự động vào một bean.
- @Service: xác định lớp được sử dụng cho tầng dịch vụ (business logic layer) (xử lý logic)
- @Repository là một chú thích cho lớp được sử dụng cho tầng lưu trữ (persistence layer) (thao tác với database)
- @Configuration là một chú thích lớp được sử dụng cho tầng cấu hình (configuration layer).
- @Bean là một chú thích cho phương thức được sử dụng để định nghĩa các bean trong các lớp được đánh dấu bằng @Configuration hoặc @Component.

Nếu có nhiều bean giống nhau trong container, @Qualifier(beanName) cho Spring biết bean nào muốn injection. Có thể sử dụng @Qualifier trên các method setter, constructor, … để chỉ định rõ ràng bean cần inject.

❖

3. Bean life cycle

@PostConstruct được đánh dấu trên một method duy nhất bên trong Bean. IoC Container hoặc ApplicationContext sẽ gọi hàm này sau khi một Bean được tạo ra và quản lý.

Tại sao không sử dụng Constructor thông thường thay cho PostConstructor?

Constructor	PostConstructor
Được chạy trước khi bean được khởi tạo => các dependency chưa được inject	Chạy sau khi bean khởi tạo thành công => Có thể sử dụng các dependency
Chạy nhiều lần khi bean khởi tạo nhiều lần	Chỉ chạy duy nhất một lần, dù bean khởi tạo nhiều lần

@PreDestroy được đánh dấu trên một method duy nhất bên trong Bean. IoC Container hoặc ApplicationContext sẽ gọi hàm này trước khi một Bean bị xóa hoặc không được quản lý nữa.

Bean life cycle được hiểu là quá trình một bean được Spring Framework tạo ra cho tới khi chết đi, sẽ có những sự kiện (event) khác nhau xảy ra

- Khi IoC Container (ApplicationContext) tìm thấy một Bean cần quản lý => khởi tạo bằng Constructor
- Inject dependencies vào Bean bằng Setter
- Các method khởi tạo khác được gọi

- Tiền xử lý trước khi @PostConstruct được gọi (implements BeanPostProcessor).
- Hàm đánh dấu @PostConstruct được gọi
- Tiền xử lý sau khi @PostConstruct được gọi (implements BeanPostProcessor).
- Bean sẵn sàng để hoạt động
- Nếu IoC Container không quản lý bean nữa hoặc bị shutdown nó sẽ gọi hàm @PreDestroy trong Bean
- Xóa Bean.

4. Run multiple environment

Environment represents the environment information for the entire spring application runtime, which contains two key elements: “profiles” and “properties”

Spring Profiles là một tính năng quan trọng của Spring Framework, cho phép quản lý và cấu hình ứng dụng dựa trên môi trường đang chạy. Một môi trường có thể tương ứng với một tập hợp cụ thể các cài đặt, tài nguyên và thiết lập

❖ Cách sử dụng Spring Profiles

- Tạo các tệp cấu hình cho từng Profile

Để sử dụng Spring Profiles, cần tạo các tệp cấu hình cho từng profile tương ứng. Các tệp cấu hình này phải được đặt tên theo mẫu application-{profile-name}.properties hoặc application-{profile-name}.yml. Ví dụ: application-local.properties, application-dev.yml.

- Cấu hình Profiles mặc định

Có thể cấu hình một profile mặc định bằng cách sử dụng thuộc tính spring.profiles.active trong tệp application.properties hoặc application.yml. ***Nếu không có profile nào được kích hoạt, Spring Boot sẽ sử dụng profile mặc định này.***

spring.profiles.active=local

- Sử dụng Profiles trong mã nguồn

Có thể sử dụng annotation @Profile để chỉ định rằng một bean hoặc một thành phần cụ thể chỉ nên được tạo ra khi một profile cụ thể được kích hoạt.

```
@Component
@Profile("local")
```

```
public class LocalDatasourceConfig {
    // Cấu hình dành cho profile "local"
}

@Component
@Profile("!local")
public class ProductionDatasourceConfig {
    // Cấu hình dành cho các profile khác ngoại trừ "local"
}
```

- Kích hoạt Profiles

Có nhiều cách để kích hoạt các profile khi chạy ứng dụng Spring Boot. Dưới đây là một số trong những cách phổ biến:

- Sử dụng JVM System Parameter: Có thể sử dụng tùy chọn -Dspring.profiles.active để chỉ định profile bạn muốn kích hoạt.
  java -jar your-app.jar -Dspring.profiles.active=local
- Sử dụng Environment Variable (Unix): Có thể đặt biến môi trường SPRING_PROFILES_ACTIVE trước khi chạy ứng dụng
  export SPRING_PROFILES_ACTIVE=local
- Sử dụng annotation @ActiveProfiles trong JUnit tests: Đối với việc kiểm tra đơn vị, bạn có thể sử dụng annotation @ActiveProfiles để kích hoạt các profile cụ thể cho các bài kiểm tra.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = YourApplication.class)
@ActiveProfiles("test")
public class YourTest {
  // Bài kiểm tra của bạn ở đây
}
```

- Chạy ứng dụng với Profiles: Khi bạn đã cấu hình và kích hoạt các profile, bạn có thể chạy ứng dụng Spring Boot với các cấu hình tương ứng với profile mà bạn đã chọn.

```
public static void main(String[] args) {
  SpringApplication application = new SpringApplication(YourApplication.class);
  ConfigurableEnvironment environment = new StandardEnvironment();
```

```
    environment.setActiveProfiles("local"); // Thay đổi profile ở đây
    application.setEnvironment(environment);
    ApplicationContext context = application.run(args);

    // Kiểm tra kết quả
    LocalDatasource localDatasource = context.getBean(LocalDatasource.class);
    System.out.println(localDatasource);
  }
```

Ví dụ khác:

```
@Configuration
public class ProfileConfiguration {
   @Bean
   @Profile("dev")
   public ProfileService profileServiceDev(){
      return new ProfileService("dev");
   }

   @Bean
   @Profile("prod")
   public ProfileService profileServiceProd(){
      return new ProfileService("prod");
   }
}
-----------------------------------------------------------------------------------------------------
public class ProfileMain {
   public static void main(String[] args) {
      AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext();
//       applicationContext.getEnvironment().setActiveProfiles("prod");
      applicationContext.register(ProfileConfiguration.class);
      applicationContext.refresh();
      System.out.println(applicationContext.getBean(ProfileService.class));
   }
}
```

https://www.springcloud.io/post/2022-02/spring-environment/

5.

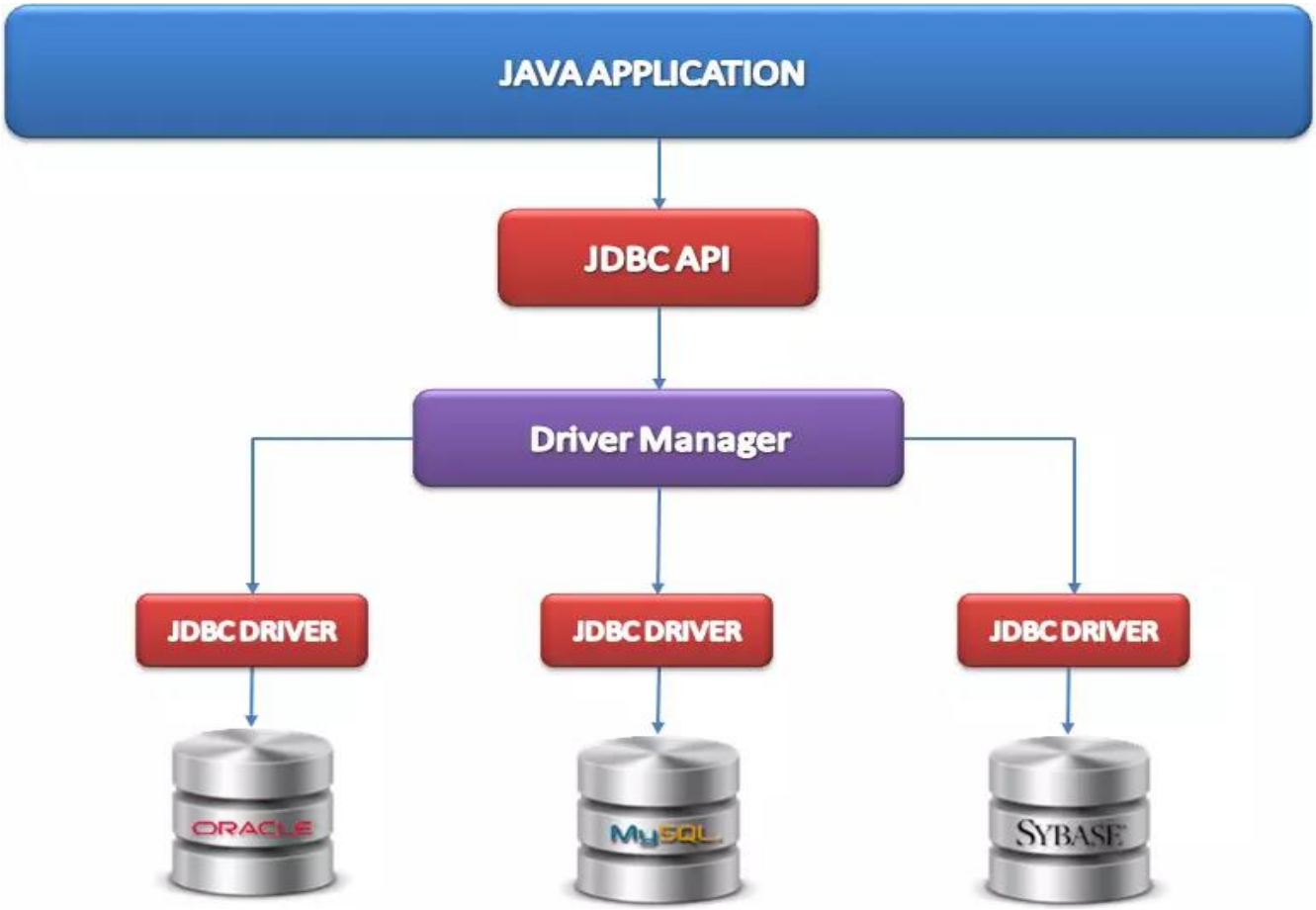
B/ Các module của Spring (khác Spring boot)

I. Hibernate

Trước khi tìm hiểu Hibernate, hãy tìm hiểu JDBC trước (hoặc có thể tìm hiểu khái quát JDBC dưới mục sau)

1. JDBC

JDBC (Java Database Connectivity) là ứng dụng mã nguồn mở cho Java, giúp ứng dụng Java thực hiện kết nối, làm việc với CSDL. Nó cho phép thực hiện các thao tác truy xuất, update dữ liệu với CSDL quan hệ bằng việc sử dụng các câu lệnh SQL (JDBC thường được thấy trong các dự án JSP/Servlet)



Ứng dụng Java sử dụng JDBC làm việc với CSDL thông qua trình tự 7 bước như sau:

- Tạo kết nối đến database
- Gửi SQL query đến database sử dụng JDBC driver tương ứng
- JDBC driver kết nối đến database
- Thực thi câu lệnh query để lấy kết quả trả về
- Gửi dữ liệu đến ứng dụng thông qua Driver Manager
- Xử lý dữ liệu trả về
- Đóng (giải phóng) kế nối đến database

Một số "vấn đề" gặp phải khi sử dụng JDBC:

- Phải lặp đi lặp lại những dòng code giống nhau trong ứng dụng chỉ để lấy dữ liệu từ database.
- Phải vất vả với việc map giữa Object Java với các table tương ứng trong database.
- Tốn nhiều công sức để thay đổi từ hệ quản trị CSDL này (MySQL) sang một hệ quản trị CSDL khác (Oracle).
- Khó khăn trong việc tạo các giao tiếp/liên hệ giữa các table, lập trình OOPs.

➔ JDBC là công cụ thô sơ nhất, mộc mạc nhất giúp kết nối CSDL trong ứng dụng Java. Và rồi Hibernate ra đời, nó mang trong mình nhiều công cụ hữu ích giúp cho việc kết nối với CSDL một cách thuận tiện, dễ dàng hơn

❖ So sánh Spring JDBC với JDBC API

Các vấn đề tồn tại với JDBC API:

- Cần viết nhiều code trước và sau thực hiện query (tạo connection, statement, đóng resultset, đóng connection…)
- Cần phải xử lý transaction

Spring JdbcTemplate cũng tiến hành giao tiếp với database thông qua cơ chế của JDBC nhưng nó cải thiện được các vấn đề trên, cung cấp các method viết query trực tiếp -> tiết kiệm thời gian…

https://stackjava.com/spring/spring-jdbc-jdbctemplate.html

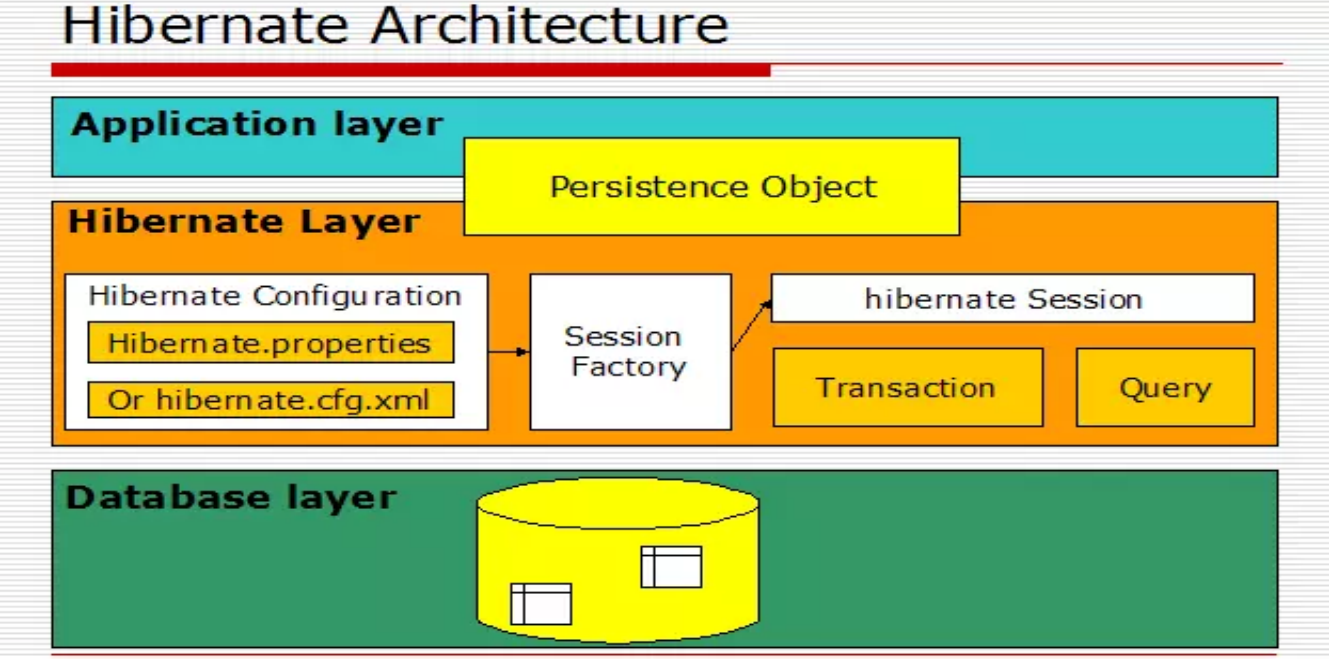
2. Hibernate

ORM (Object-Relational Mapping) là một kỹ thuật được sử dụng trong lập trình để ánh xạ giữa object trong mã nguồn và các table trong cơ sở dữ liệu. Nó giúp chúng ta làm việc với cơ sở dữ liệu một cách trừu tượng hóa, tức là thay vì sử dụng truy vấn SQL trực tiếp, chúng ta sử dụng các đối tượng Java để thực hiện thao tác cơ sở dữ liệu.

Lợi ích của ORM:

- Trừu tượng hóa cơ sở dữ liệu: Có thể làm việc với cơ sở dữ liệu mà không cần quan tâm đến chi tiết cụ thể của nó => giúp tách biệt logic ứng dụng và logic truy cập dữ liệu.
- Giảm việc viết SQL thủ công: Không cần phải viết nhiều truy vấn SQL thủ công, giúp giảm thiểu lỗi và tiết kiệm thời gian.
- Dễ bảo trì và mở rộng

Hibernate là một thư viện ORM (Object Relational Mapping) mã nguồn mở giúp lập trình viên viết ứng dụng Java có thể map các objects (POJO) với hệ quản trị cơ sở dữ liệu quan hệ, và hỗ trợ thực hiện các khái niệm lập trình hướng đối tượng với cơ dữ liệu quan hệ



Persistence object chính là các POJO object map với các table tương ứng của cơ sở dữ liệu quan hệ. Nó như là những "thùng xe" chứa dữ liệu từ ứng dụng để ghi xuống database, hay chứa dữ liệu tải lên ứng dụng từ database.

Session Factory là một interface giúp tạo ra session kết nối đến database bằng cách đọc các cấu hình trong Hibernate configuration. Mỗi một database phải có một session factory. Ví dụ nếu ta sử dụng MySQL, và Oracle cho ứng dụng Java của mình thì ta cần có một session factory cho MySQL, và một session factory cho Oracle.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://192.168.10.13:3306/data
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
```

```
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.current_session_context_class">thread</property>
    </session-factory>
</hibernate-configuration>
```

Hibernate Session: Mỗi một đối tượng session được Session factory tạo ra sẽ tạo một kết nối đến database. Transaction là transaction đảm bảo tính toàn vẹn của phiên làm việc với cớ sở dữ liệu. Tức là nếu có một lỗi xảy ra trong transaction thì tất cả các tác vụ thực hiện sẽ thất bại. Query Hibernate cung cấp các câu truy vấn HQL (Hibernate Query Language) tới database và map kết quả trả về với đối tượng tương ứng của ứng dụng Java.

3. Lý do lựa chọn hibernate thay vì JDBC?

❖ Object mapping

- Với JDBC: Phải map các trường trong bảng với các thuộc tính của Java object một cách "thủ công".
- Với Hibernate: Hỗ trợ map một cách "tự động" thông qua các file cấu hình map XML hay sử dụng các annotation (@Entity, @Table, @Column, …)

❖ HQL

Hibernate cung cấp các câu lệnh truy vấn tương tự SQL, HQL của Hibernate hỗ trợ đầy đủ các truy vấn đa hình như, HQL "hiểu" các khái niệm như kế thừa (inheritance), đa hình (polymorphysm), và liên kết (association)

❖ Database Independent

Code sử dụng Hibernate là độc lập với hệ quản trị cơ sở dữ liệu, nghĩa là không cần thay đổi câu lệnh HQL khi chuyển đổi từ hệ quản trị CSDL MySQL sang Oracle, hay các hệ quản trị CSDL khác... Do đó rất dễ để thay đổi CSDL quan hệ, đơn giản bằng cách thay đổi thông tin cấu hình hệ quản trị CSDL trong file cấu hình

❖ Minimize Code Changes

Khi ta thay đổi (thêm) cột vào bảng:

- Với JDBC:
  - Thêm thuộc tính vào POJO class.

- Thay đổi method chứa câu truy vấn "select", "insert", "update" để bổ sung cột mới. Có thể có rất nhiều method, nhiều class chứa các câu truy vấn như trên.

- Với Hibernate: Chỉ cần thêm thuộc tính vào POJO class. Cập nhật Hibernate XML mapping file để thêm map column – property → chỉ thay đổi duy nhất 2 file trên.

❖ Lazy Loading

Với những ứng dụng Java làm việc với CSDL lớn hàng trăm triệu bản ghi, việc có sử dụng Lazy loading trong truy xuất dữ liệu từ database mang lại lợi ích rất lớn.

Ví dụ: Bảng user quan hệ 1-n với bảng document. Trong trường hợp này, User là class cha và Document là class con. Giả sử bảng document chứa 1 lượng lớn dữ liệu. Mỗi khi lấy thông tin user và không cần thiết phải lấy document tương ứng từ database nhằm để ứng dụng không bị chậm vì phải mất nhiều bộ nhớ để chứa toàn bộ document của user, thì áp dụng Lazy loading với FetchType.LAZY

Công dụng của FetchType.LAZY:

- Trì hoãn tải dữ liệu: Các dữ liệu liên quan đến mối quan hệ sẽ không được tải từ cơ sở dữ liệu ngay lập tức mà chỉ được tải khi cần thiết.
- Tăng hiệu suất ứng dụng: Tránh tải dữ liệu không cần thiết từ cơ sở dữ liệu, giảm tải cho máy chủ và tăng hiệu suất ứng dụng.

❖ Loại bỏ Try-Catch Blocks

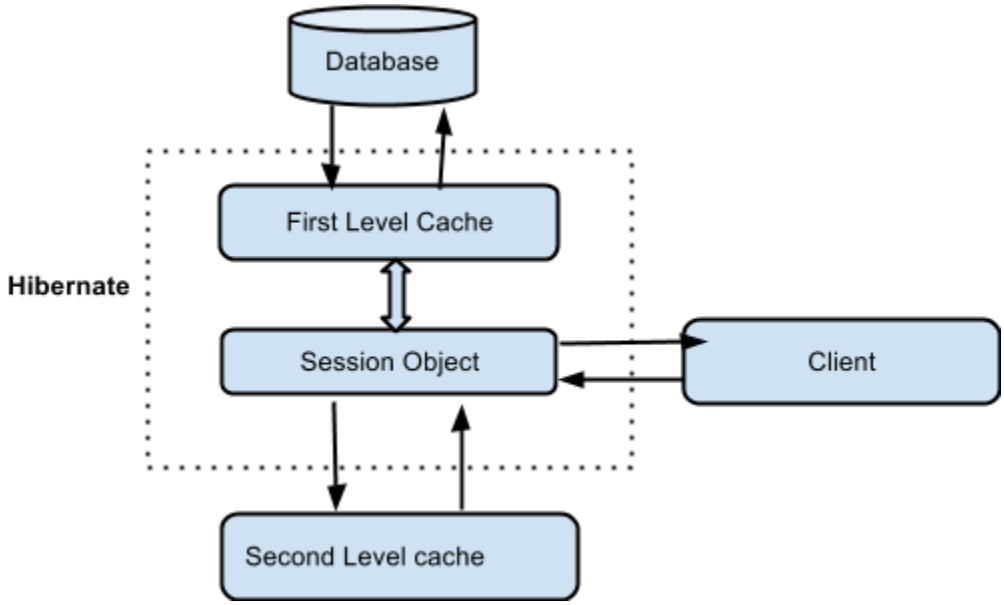
- Với JDBC, nếu lỗi xảy khi tao tác với database thì sẽ có exception SQLexception bắn ra. Bởi vậy phải sử dụng try-catch block để xử lý ngoại lệ.
- Với Hibernate: Nó override toàn bộ JDBC exception thành Uncheck exeption, và không cần viết try-catch trong code nữa.

❖ Quản lý commit/rollback Transaction

Transaction là nhóm các hoạt động (với database) của một tác vụ. Nếu một hoạt động không thành công thì toàn bộ tác vụ không thành công.

- Với JDBC, lập trình viên phải chủ động thực hiện commit khi toàn bộ hoạt động của tác vụ thành công, hay phải rollback khi có một hoạt động không thành công để kết thúc tác vụ.
- Với Hibernate thì không cần quan tâm đến commit hay rollback, Hibernate đã tự quản lý rồi

❖ Hibernate Caching



Hibernate cung cấp 1 cơ chế bộ nhớ đệm giúp giảm số lần truy cập vào database của ứng dụng và tăng performance đáng kể cho ứng dụng. Hibernate lưu trữ các đối tượng trong session khi transation được kích hoạt. Khi một query được thực hiện liên tục, giá trị được lưu trữ trong session được sử dụng lại. Khi một transaction mới bắt đầu, dữ liệu được lấy lại từ database và được lưu trữ session.

4. Connection pooling

Đặt vấn đề: Thiết lập kết nối database rất tốn tài nguyên và đòi hỏi nhiều chi phí. Hơn nữa, trong môi trường đa luồng, việc mở và đóng nhiều kết nối thường xuyên và liên tục ảnh hưởng nhiều đến performance và tài nguyên. => connection pooling

Connection pool: là kỹ thuật cho phép tạo và duy trì 1 tập các kết nối dùng chung nhằm tăng hiệu suất cho ứng dụng bằng cách sử dụng lại các kết nối khi có yêu cầu thay vì việc tạo kết nối mới.

Connection Pool Manager (CPM) là trình quản lý vùng kết nối. Khi ứng dụng start thì Connection pool tạo ra một vùng kết nối, trong vùng đó có các kết nối do chúng ta tạo ra sẵn. Như vậy, khi có một request đến thì CPM kiểm tra xem có kết nối nào đang rảnh không? Nếu có nó sẽ dùng kết nối đó, nếu không thì nó sẽ đợi cho đến khi có kết nối nào đó rảnh hoặc kết nối khác bị timeout. Kết nối sau khi sử dụng sẽ không đóng lại ngay mà sẽ được trả về CPM để dùng lại khi được yêu cầu trong tương lai.

Ưu điểm:

- Hiệu năng cao và tối ưu hóa tài nguyên: Giúp tăng hiệu suất và tối ưu hóa tài nguyên bằng cách tái sử dụng các kết nối đã được thiết lập, thay vì phải tạo mới mỗi lần cần truy cập database

- Quản lý kết nối hiệu quả: Cung cấp cơ chế quản lý các kết nối dễ dàng và tin cậy, theo dõi trạng thái (không sử dụng, đang sử dụng) và các thiết lập cấu hình.
- Tính sẵn sàng và khả năng mở rộng: Giúp ứng dụng có khả năng xử lý tải cao hơn bằng cách cung cấp các kết nối sẵn có từ pool khi có nhiều yêu cầu đến cùng một lúc.

Nhược điểm:

- Cấu hình phức tạp
- Tiêu tốn bộ nhớ: Yêu cầu nhiều bộ nhớ để duy trì pool các kết nối => ảnh hưởng đến tài nguyên hệ thống
- Khả năng phục hồi: Nếu có sự cố xảy ra với cơ sở dữ liệu, việc phục hồi các kết nối trong pool có thể tiêu tốn thời gian và tài nguyên hơn so với việc tạo kết nối mới.
- Cập nhật cấu hình: Việc cập nhật cấu hình connection pooling (như thay đổi số lượng kết nối tối đa) có thể gây ảnh hưởng đến ứng dụng và yêu cầu sự chú ý cẩn thận.

❖ JDBC Connection Pooling

JDBC Connection Pooling tuân thủ theo Connection Pooling. Sau khi ứng dụng được start, một Connection Pool object được tạo, các Connection object sau này được tạo sẽ được quản lý bởi pool này.

Có nhiều thư viện hỗ trợ JDBC Conection Pooling: C3P0, Apache Commons DBCP, HikariCP, ...

Ví dụ:

```
public class DBCP2Source {
   private static Logger logger = LogManager.getLogger(DBCP2Source.class);
   private static BasicDataSource dataSource = new BasicDataSource();

   static { // run only once
      dataSource.setUrl(DBConfiguration.DB_URL);
      dataSource.setUsername(DBConfiguration.USER_NAME);
      dataSource.setPassword(DBConfiguration.PASSWORD);
      dataSource.setMinIdle(1); //duy trì tối thiểu
      dataSource.setMaxIdle(2); //duy trì tối đa
      dataSource.setMaxTotal(2); //tối đa có thể có
   }

   public static Connection getConnection() throws SQLException {
```

```
        Connection connection = dataSource.getConnection();
        logger.info("+ Num of Idle Connections:: " + dataSource.getNumIdle()); //free
        logger.info("+ Num of Busy Connections: " + dataSource.getNumActive()); //busy
        return connection;
    }
}

public class UserService {
    public List<User> getAll() {
        try(Connection connection = DBCP2Source.getConnection()) {
            Statement statement = connection.createStatement();
            ResultSet rs = statement.executeQuery("SELECT * FROM user");
            Thread.sleep(2000);

            while (rs.next()) {
                // ... xử lý
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return users;
    }
}
```

https://topdev.vn/blog/gioi-thieu-jdbc-connection-pool/#connection-pooling-la-gi

https://shareprogramming.net/jdbc-connection-pooling-la-gi-trien-khai-jdbc-connection-pooling-trong-java/#Tai_sao_can_den_Connecion_Pooling

- ❖ Connection Pooling trong JPA

5. Cài đặt, sử dụng Hibernate

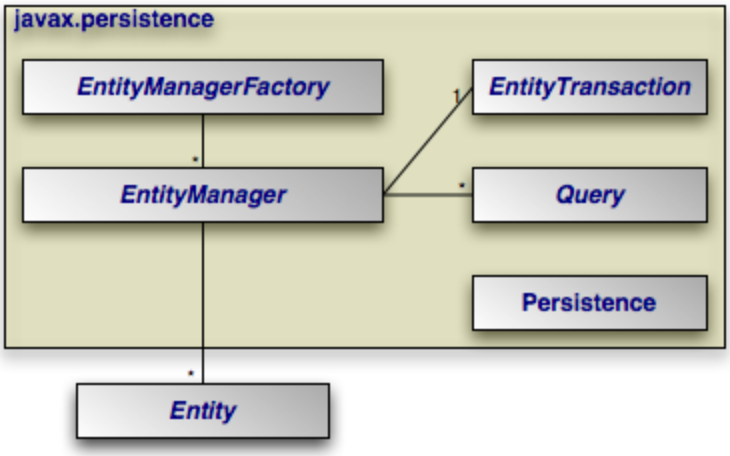
II. Spring Data JPA

1. JPA

JPA (Java Persistence API) là 1 interface lập trình ứng dụng Java, nó mô tả cách quản lý các mối quan hệ dữ liệu trong ứng dụng sử dụng Java Platform.

JPA cung cấp một mô hình POJO persistence cho phép ánh xạ các table/các mối quan hệ giữa các table trong database sang các class/mối quan hệ giữa các object. Vì vậy, thay vì truy vấn table hay các column bằng syntax của SQL, ta sẽ truy vấn trực tiếp trên các class, các field của class bằng JPA Query Language (JPQL). Với cách này, ta sẽ không cần quan tâm tới việc đang dùng loại database nào (mỗi loại database có thể có syntax khác nhau), dữ liệu database ra sao, …

JPA đi kèm với một tập hợp các annotations như @Table, @Entity, @Id và @Column.



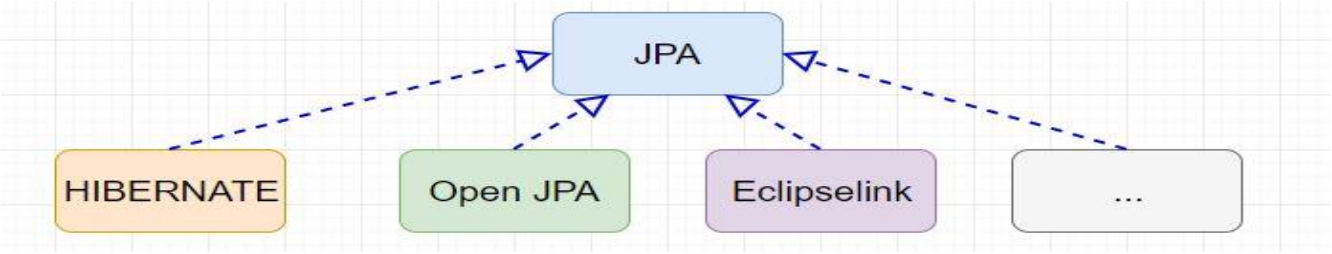
Entity là các đối tượng thể hiện tương ứng 1 table trong cơ sở dữ liệu. Khi lập trình, entity thường là các class POJO đơn giản, chỉ gồm các method getter, setter.

EntityManager là một interface cung cấp các API cho việc tương tác với các Entity như Persist (lưu một đối tượng mới), merge (cập nhật một đối tượng), remove (xóa 1 đối tượng).

EntityManagerFactory được dùng để tạo ra một thể hiện của EntityManager

➔ JPA là 1 tập các interface, còn Hibernate implements các interface ấy (nghĩa là JPA chỉ định nghĩa các đặc tả cần thiết và không có code hiện thực từ những đặc tả đó)

Ngoài Hibernate ra, còn có 1 số framework khác như Open JPA, Eclipselink cũng thực hiện implements JPA nhưng Hibernate được sử dụng phổ biến hơn



JPA sử dụng JPQL (Java Persistence Query Language), Hibernate sử dụng HQL (Hibernate Query Language). JPQL và HQL đều là các ngôn ngữ truy vấn dựa trên SQL, trong đó, JPQL là một phần của tiêu chuẩn JPA, do đó nó được hỗ trợ bởi tất cả các triển khai JPA. HQL là một ngôn ngữ truy vấn riêng biệt của Hibernate, do đó nó chỉ được hỗ trợ bởi Hibernate.

❖ JPQL

- JPQL cho phép định nghĩa các câu query dựa trên các entity chứ không dựa vào tên các cột, các bảng trong database.
- Cấu trúc và cú pháp của JPQL thì tương tự như cấu trúc và cú pháp của câu SQL.

Điều này giúp dễ dàng định nghĩa các câu query sử dụng JPQL nhưng nên nhớ rằng: mặc dù định nghĩa các câu query sử dụng các entity nhưng trong thực tế, lúc chạy, Hibernate hay bất kỳ thư viện nào implement JPA đều transform những câu query đó sang những câu SQL dành cho database với tên cột, tên bảng của database đó

Ví dụ:

```
public interface UserRepository extends JpaRepository<User, Long> {
    // Khi được gắn @Query, thì tên của method không còn tác dụng nữa
    // Đây là JPQL
    @Query("select u from User u where u.emailAddress = ?1")
    User myCustomQuery(String emailAddress);

    // Đây là Native SQL
    @Query(value = "select * from User u where u.email_address = ?1", nativeQuery = true)
    User myCustomQuery2(String emailAddress);

    // JPQL
    @Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")
    User findUserByNamedParams(@Param("status") Integer status, @Param("name") String name);

    // Native SQL
    @Query(value = "SELECT * FROM Users u WHERE u.name = :name", nativeQuery = true)
    User findUserByNamedParamsNative( @Param("name") String name);
}
```

❖ Criteria API

Biết JPQL có thể thực hiện truy vấn đầy đủ chỉ với 1 câu lệnh, tuy nhiên, lại khó tùy biến hay tái sử dụng, khó kiểm soát lỗi.

Vì vậy Criteria API được sinh ra, cho phép xây dựng câu lệnh một cách dynamic, không bị hardcode trong một String và có thể tái sử dụng.

Ví dụ:

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("yourPersistenceUnitName");
EntityManager em = entityManagerFactory.createEntityManager();
CriteriaBuilder builder = em.getCriteriaBuilder();


CriteriaQuery<Office> criteriaQuery = builder.createQuery(Office.class);
Root<Office> root = criteriaQuery.from(Office.class);


// SELECT o FROM Office o WHERE o.city = 'hanoi'
criteriaQuery.select(root).where(builder.equal(root.get("city"), "hanoi"));
TypedQuery<Office> typedQuery = em.createQuery(criteriaQuery);
List<Office> results = typedQuery.getResultList();
```

CriteriaBuilder: Giúp tạo ra đối tượng chứa câu lệnh truy vấn CriteriaQuery và cung cấp cơ số các phép biến đổi, phép logic, điều kiện cho câu lệnh (and, or, not, avg, greater than, ...)

CriteriaQuery: Khai báo đối tượng muốn lấy ra sau khi thực hiện query

❖ Specification

2. Spring Data JPA

Nếu sử dụng JPA cùng với Spring framework trong dự án của thì hãy sử dụng Spring Data JPA.

Spring Data JPA là một module nhỏ trong một project lớn gọi là Spring Data project (Spring Data đã wrapper Hibernate để tạo thành Spring Data JPA). Mục đích của Spring Data project là giảm thiểu các đoạn code lặp đi lặp lại liên quan đến phần thao tác với các hệ thống quản trị data khi phát triển các ứng dụng có sử dụng Spring framework. Ngoài Spring Data JPA hỗ trợ cho JPA giảm thiểu code để truy cập và thao tác với các hệ thống quản trị cơ cở dữ liệu, còn có Spring Data JDBC (cũng giống như

Spring Data JPA), Spring Data LDAP (hỗ trợ Spring LDAP), Spring Data MongoDB (hỗ trợ cho MongoDB), …

Để đạt được mục đích giảm thiểu code như trên, Spring Data định nghĩa một interface chính tên là Repository nằm trong module Spring Data Common, module này sẽ được sử dụng cho tất cả các module còn lại trong Spring Data project. Nội dung của interface này đơn giản như sau:

```
package org.springframework.data.repository;
import org.springframework.stereotype.Indexed;
@Indexed
public interface Repository<T, ID> {
}
```

Vì interface này đơn giản như vậy nên sẽ có nhiều interface khác extend từ interface repository tuỳ thuộc vào module sử dụng. Và interface CrudRepository là một interface duy nhất extend interface Repository mà Spring Data JPA đang sử dụng.

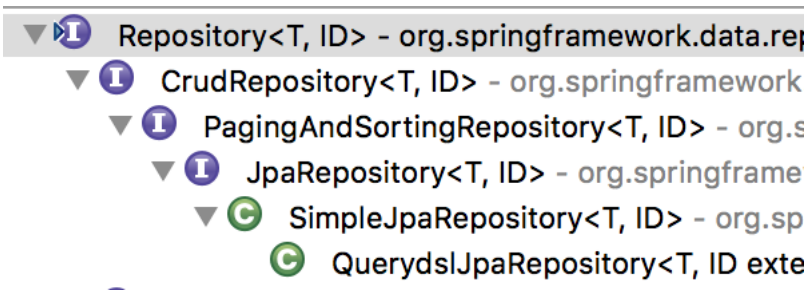
Interface CrudRepository với ý nghĩa create, read, update, delete cho phép thực hiện các thao tác cơ bản đến với các hệ thống data (không chỉ là database).

Để hỗ trợ việc phân trang và sắp xếp cho Spring Data JPA, thì còn có một interface khác là PagingAndSortingRepository.

Tất cả các interface trên đều nằm trong module Spring Data Common.

Trong Spring Data JPA, ở đây, chỉ có duy nhất một interface là JpaRepository kế thừa interface PagingAndSortingRepository. Với việc extend từ interface PagingAndSortingRepository, ta cũng có thể hình dung là Spring Data JPA có thể giúp giảm thiểu code cho các thao tác liên quan đến database.

Cấu trúc extend cho những interface:



Tham khảo:

https://huongdanjava.com/vi/tong-quan-ve-spring-data-jpa.html

https://huongdanjava.com/vi/jpa-va-spring-framework.html

https://spring.io/projects/spring-data

3. Cascade

Cascade là một tính năng giúp quản lý trạng thái của các đối tượng trong một mối quan hệ một cách tự động.

Ví dụ:

```
@Entity
@Table(name = "company")
public class Company {
  //...

  @OneToMany(fetch = FetchType.LAZY, mappedBy = "company", cascade = CascadeType.REMOVE)
  private Set<Employee> listEmployee = new HashSet<>();
  //...

}
```

Nghĩa là, khi xóa Company thì Set<Employee> cũng bị xóa theo

Một số loại cascade khác:

- CascadeType.PERSIST: Đồng bộ hóa các hoạt động thêm dữ liệu (INSERT) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.MERGE: Đồng bộ hóa các hoạt động cập nhật dữ liệu (UPDATE) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.REMOVE: Đồng bộ hóa các hoạt động xóa dữ liệu (DELETE) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.REFRESH: Đồng bộ hóa các hoạt động làm mới dữ liệu (REFRESH) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.DETACH: Đồng bộ hóa các hoạt động tách rời dữ liệu (DETACH) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.ALL: Bao gồm tất cả các loại cascade đã đề cập trên (PERSIST, MERGE, REMOVE, REFRESH, DETACH).

4. Fetch

```
@Entity
@Table(name = "company")
public class Company {
  //...
  @OneToMany(fetch = FetchType.LAZY, mappedBy = "company")
  private List<Employee> listEmployee = new ArrayList<>();
}
```

fetch = FetchType.LAZY

Khi find, select đối tượng Company từ database thì nó sẽ không lấy các đối tượng Employee liên quan. Nhưng bên trong transaction, khi gọi company.getListEmployee() thì nó vẫn có dữ liệu và khi kết thúc transaction listEmployee sẽ chứa các employee liên quan.

Ưu điểm: tiết kiệm thời gian và bộ nhớ

Nhược điểm: gây ra lỗi LazyInitializationException, khi muốn lấy các đối tượng liên quan phải mở transaction 1 lần nữa để query

fetch = FetchType.EAGER

Khi bạn find, select đối tượng Company từ database thì tất cả các đối tượng Employee liên quan sẽ được lấy ra và lưu vào listEmployee. Do đó khi kết thúc transaction, listEmployee sẽ có chứa các đối tượng Employee của Company đó.

Ưu điểm: có thể lấy luôn các đối tượng liên quan, xử lý đơn giản, tiện lợi

Nhược điểm: tốn nhiều thời gian và bộ nhớ khi select, dữ liệu lấy ra bị thừa, không cần thiết.

NOTE:

Với annotation @ManyToOne và @OneToOne thì fetchType mặc định là EAGER vì nó chỉ lấy ra nhiều nhất 1 đối tượng liên quan nên không ảnh hưởng gì tới performance

Với annotation @ManyToMany và @OneToMany thì fetchType mặc định là LAZY vì nó lấy ra rất nhiều đối tượng liên quan dẫn tới làm giảm hiệu năng, tốn bộ nhớ

5. Query DSL

Tham khảo:

https://www.baeldung.com/rest-api-search-language-spring-data-querydsl

6. Annotations

❖ @OneToOne

Dùng để biểu diễn quan hệ 1-1

Ví dụ: 1 User có 1 Address duy nhất

Khi đó, từ User ta có thể lấy ra Address và từ Address ta có thể lấy ra User

```
Address address = new Address ("Address 1");
User user  = new User();
user.setAddress(address);
user.save(); // như vậy ta sẽ lưu dữ liệu xuống 2 bảng User và Address
```

❖ @ManyToOne và @OneToMany

@ManyToOne cho phép bạn để ánh xạ cột khóa ngoại (foreign key) trong thực thể con kết nối để đối tượng con có thể có một đối tượng thực thể tham chiếu đến thực thể cha của nó.

Cho thuận tiện, để tận dụng các chuyển đổi trạng thái thực thể, nhiều lập trình viên chon ánh xạ thực thể con như một tập hợp (collection) trong đối tượng cha, vì mục đích nay JPA cung cấp chú thích @OneToMany.

```
public class Person {
   @Id
   private Long id;

   private String name;

   @ManyToOne
   @JoinColumn(name = "address_id")
   private Address address;
}
```

```
public class Address {
   @Id
   private Long id;

   private String city;
   private String province;

   @OneToMany(mappedBy = "address")
   private Collection<Person> persons;
}
```

❖ @ManyToMany

Dùng để biểu diễn quan hệ n - n

```
public class Person {
   @Id
```

```
public class Address {
   @Id
```

```
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "persons")
    private Collection<Address> addresses;
}
```

```
    private Long id;

    private String city;
    private String province;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
    name = "address_person",
    joinColumns = @JoinColumn(name = "address_id"),
    inverseJoinColumns = @JoinColumn(name = "person_id")
    )
    private Collection<Person> persons;
}
```

❖ @Column

@Column là một annotation được sử dụng để ánh xạ một trường (field) của một entity (thực thể) vào một cột trong cơ sở dữ liệu. Annotation này cung cấp các thuộc tính để tùy chỉnh cách cột được tạo trong cơ sở dữ liệu.

Fetch:

updatable:

Thuộc tính updatable trong @Column annotation của JPA được sử dụng để chỉ định xem cột tương ứng có thể được sử dụng trong các câu lệnh SQL UPDATE hay không. Khi đặt giá trị của updatable là false (*@Column(name = "created_date", updatable = false*) , nghĩa là cột đó sẽ không được cập nhật khi thực hiện các thao tác cập nhật trên entity tương ứng. Điều này hữu ích khi bạn muốn loại bỏ khả năng cập nhật một số cột trong trường hợp nào đó.

columnDefinition:

Được sử dụng để chỉ định kiểu dữ liệu và các thuộc tính khác cho cột tương ứng trong cơ sở dữ liệu. Thuộc tính này cho phép bạn mô tả cụ thể các đặc tính của cột, chẳng hạn như kiểu dữ liệu, độ dài, ràng buộc, và các thuộc tính khác tùy thuộc vào cơ sở dữ liệu mà bạn đang sử dụng.

Ví dụ, bạn có thể sử dụng columnDefinition để chỉ định kiểu dữ liệu VARCHAR và độ dài 100 cho một cột trong MySQL như sau:

@Column(name = "first_name", columnDefinition = "VARCHAR(100)")

private String firstName;

Trong trường hợp này, khi entity được tạo, cột first_name trong cơ sở dữ liệu sẽ được tạo với kiểu dữ liệu VARCHAR và độ dài 100.

Lưu ý rằng việc sử dụng columnDefinition có thể là cách cụ thể cho một cơ sở dữ liệu cụ thể và có thể không di động giữa các cơ sở dữ liệu khác nhau. Nếu bạn muốn ứng dụng của bạn có tính di động cao hơn, hãy xem xét sử dụng các annotation khác như @Length hoặc @Size để chỉ định độ dài hoặc kích thước của trường mà không liên quan trực tiếp đến cơ sở dữ liệu cụ thể.

- **@Enumerated**

@Enumerated là 1 annotation của JPA, sử dụng để chỉ định cách mà các giá trị của một enum sẽ được mapping vào db. Thường được sử dụng khi muốn lưu trữ giá trị của enum dưới dạng chuỗi (STRING) thay vì giá trị số (ORDINAL).

@Enumerated(EnumType.STRING)

private ProductType type;

ProductType là một enum và type sẽ được mapping vào column trong db có kiểu STRING. Khi lấy dữ liệu từ db và mapping vào Java object, JPA sẽ chuyển đổi giá trị từ chuỗi thành giá trị enum tương ứng.

Việc sử dụng EnumType.STRING thay vì EnumType.ORDINAL, đảm bảo tính rõ ràng và ổn định trong db. Khi sử dụng ORDINAL, mọi thay đổi trong enum có thể làm thay đổi thứ tự của các giá trị enum, gây khó khăn trong việc duy trì và cải thiện độ tin cậy của dữ liệu. STRING giúp giải quyết vấn đề này, tuy nhiên có thể làm cho db có kích thước lớn

- **@EmbeddedId**

Là 1 annotation của JPA, sử dụng để xác định 1 field hoặc 1 class nhúng (embedded) làm primary key cho 1 đối tượng entity. Nó thường được sử dụng trong các trường hợp mà khóa chính của đối tượng bao gồm nhiều field hoặc cần được ánh xạ vào 1 class đặc biệt để quản lý.

Khi sử dụng @EmbeddedId, cần tạo ra 1 class mới đại diện cho primary key, và sử dụng @Embeddable trên class này. Trong class đó, sẽ định nghĩa các field mà ta muốn sử dụng.

```
@Embeddable
public class EmployeeId implements Serializable {
    private Long departmentId;
    private Long employeeNumber;
    // Constructors, getters, setters, and other methods
}
```

```
@Entity
public class Employee {
    @EmbeddedId
    private EmployeeId employeeId;
    private String firstName;
    private String lastName;
    // Constructors, getters, setters, and other
methods
}
```

❖

III. Spring Data Redis

1. Tổng quan về Caching

Caching là một kỹ thuật trong lập trình, được sử dụng để lưu trữ tạm thời các dữ liệu hoặc các tài nguyên có thể được truy cập một cách nhanh chóng. Mục tiêu của caching là giảm thời gian truy cập dữ liệu, tăng cường hiệu suất và giảm gánh nặng cho hệ thống.

Khi một ứng dụng yêu cầu dữ liệu, trước hết nó sẽ kiểm tra xem dữ liệu đã được lưu trong bộ nhớ cache hay chưa. Nếu dữ liệu đã tồn tại trong cache, ứng dụng sẽ lấy nhanh chóng từ đó thay vì phải truy cập vào nơi lưu trữ gốc (cơ sở dữ liệu, API, hoặc tệp tin). Nếu dữ liệu không có trong cache, ứng dụng sẽ lấy từ nơi lưu trữ gốc và sau đó lưu vào cache để sử dụng cho các lần truy cập sau.

2. Các phương pháp caching dữ liệu

❖ **Spring Cache Abstraction**

Sử dụng các annotations (@Cacheable, @CacheEvict, @CachePut, @Caching) để quản lý cache một cách trừu tượng mà không cần quan tâm đến việc thực hiện cụ thể.

Nó không phải là một trình quản lý cache thực tế, mà là một API trừu tượng giúp bạn dễ dàng tích hợp với các trình quản lý cache khác như EhCache, Caffeine, Hazelcast, Guava, Redis, ...

@Cacheable: Đánh dấu một phương thức rằng kết quả của nó có thể được cache. Khi phương thức được gọi, Spring sẽ kiểm tra xem kết quả của phương thức đó đã được cache chưa. Nếu có, kết quả từ cache sẽ được trả về thay vì thực hiện phương thức.

@CacheEvict: Dùng để loại bỏ dữ liệu khỏi cache. Bạn có thể cấu hình nó để xóa một mục cụ thể khỏi cache hoặc để xóa tất cả các mục trong một cache cụ thể.

@CachePut: Luôn thực hiện phương thức và kết quả của nó sẽ được đưa vào cache theo khóa chỉ định. Điều này hữu ích khi bạn muốn cập nhật cache với một kết quả mới.

@Caching: Được sử dụng để nhóm nhiều chú thích caching (kết hợp @Cacheable, @CachePut, và @CacheEvict) trên một phương thức duy nhất.

@CacheConfig: Dùng để chia sẻ một số thuộc tính cache giữa các phương thức trong một lớp.

Spring Cache rất linh hoạt vì không chỉ có thể sử dụng các trình quản lý cache in-memory như EhCache hay Caffeine mà còn có thể tích hợp với các hệ thống cache phân tán như Redis hay Hazelcast.

Cấu hình của cache có thể được quản lý thông qua các file cấu hình như application.properties, application.yml hoặc bằng cách sử dụng Java Configuration (@Configuration)

Mặc dù được thiết kế để làm việc với bất kỳ trình quản lý cache nào nhưng không yêu cầu phải tích hợp với một trình quản lý cụ thể.

Khi bạn không cung cấp một cấu hình cache cụ thể, Spring sẽ tạo ra một SimpleCacheMemory với một ConcurrentMapCache mặc định, là một cách đơn giản để thực hiện caching in-memory mà không cần bất kỳ sự tích hợp đặc biệt nào

ConcurrentMapCache là một cách tiện lợi để bắt đầu với caching trong Spring nếu bạn không có nhu cầu phức tạp và không cần một giải pháp caching mạnh mẽ như Caffeine, Redis, EhCache, ... Đây là một cách tốt để thêm caching vào ứng dụng trong các trường hợp sử dụng đơn giản, không yêu cầu tính năng đầy đủ của một trình quản lý cache chuyên nghiệp.

Sử dụng dependency: spring-boot-starter-cache

- **In-Memory Cache**

Là một hình thức caching trong đó dữ liệu được lưu trữ trực tiếp trong bộ nhớ chính (RAM) của máy chủ => giúp việc truy xuất dữ liệu nhanh hơn nhiều so với việc đọc từ ổ đĩa cứng hoặc một nguồn dữ liệu từ xa như cơ sở dữ liệu hoặc dịch vụ web.

Tuy nhiên, RAM có kích thước hạn chế và dữ liệu trong cache thường bị mất khi ứng dụng hoặc máy chủ bị tắt => cần phải có chiến lược để quản lý việc đặt vào và loại bỏ khỏi cache (như eviction policies), cũng như để đồng bộ hóa cache trong một hệ thống phân tán.

Công cụ phổ biến để thực hiện In-Memory Cache:

- HashMap hoặc ConcurrentHashMap: Là cách đơn giản nhất nhưng không cung cấp các tính năng quản lý cache tự động.
- Caffeine: Một thư viện caching nhanh chóng, mạnh mẽ với các cơ chế quản lý cache như evictions, loading, và computation.
- Guava: Trước khi có Caffeine, Guava là lựa chọn phổ biến cho in-memory cache với các tính năng tương tự nhưng không hiệu suất cao như Caffeine.

Ưu điểm:

- Đơn giản và dễ triển khai
- Thích hợp cho các ứng dụng nhỏ và không phân tán.

Nhược điểm:

- Không hỗ trợ tính năng như time-to-live, evictions tự động, và khả năng mở rộng.
- Không thích hợp cho các ứng dụng có kích thước dữ liệu lớn hoặc đòi hỏi khả năng mở rộng.

=> nên sử dụng cho ứng dụng nhỏ và không phân tán

❖ **Distributed Cache**

Là một hệ thống cache được thiết kế để hoạt động trên nhiều máy chủ hoặc nút trong một mạng. Mục đích chính là để tăng tốc độ truy cập dữ liệu bằng cách lưu trữ bản sao của dữ liệu thường xuyên được truy cập hoặc tính toán tốn kém trên nhiều địa điểm trong mạng.

Các đặc điểm:

- Phân tán và đồng bộ: Dữ liệu được phân tán giữa nhiều nút hoặc máy chủ và thường được đồng bộ hóa hoặc sao chép giữa các nút để đảm bảo tính nhất quán của dữ liệu.
- Tăng tốc độ và giảm độ trễ: Cung cấp dữ liệu từ cache gần với vị trí của người dùng giúp giảm độ trễ và tăng tốc độ truy cập dữ liệu.
- Tăng khả năng mở rộng: Hỗ trợ khả năng mở rộng ngang, cho phép hệ thống mở rộng bằng cách thêm nhiều nút cache hơn mà không ảnh hưởng đến hiệu suất.

- Tăng khả năng chịu lỗi: Khi mất mát dữ liệu ở một nút nhưng không làm mất mát dữ liệu hoặc làm gián đoạn dịch vụ, vì dữ liệu có thể được truy cập từ các nút khác.

Các công cụ phổ biến:

- Redis: Một cơ sở dữ liệu key-value nhanh chóng, hỗ trợ các cấu hình như cache phân tán, với khả năng lưu trữ dữ liệu trong bộ nhớ và cung cấp các cấu trúc dữ liệu phức tạp.
- Memcached: Một hệ thống caching phân tán nhẹ, dễ dàng cấu hình, tập trung vào việc lưu trữ dữ liệu key-value đơn giản trong bộ nhớ.
- Hazelcast: Một in-memory data grid, cung cấp các cấu trúc dữ liệu phân tán và hỗ trợ transaction, được sử dụng để tạo ra các hệ thống có khả năng chịu lỗi và mở rộng cao.

Về bản chất, các công cụ này cũng có thể được sử dụng cho mục đích In-memory cache. Tuy nhiên khi nhắc đến việc caching trên các hệ thống phân tán, chúng sẽ là những lựa chọn hàng đầu. Khi mà các công cụ In-memory cache không đáp ứng được hoặc khó để triển khai.

❖ **Một vài phương pháp caching dữ liệu khác**

- Cache with Database
- Application Server Caching
- HTTP Caching
- Web Browser Caching
- CDN (Content Delivery Network)

3. Spring Data Redis?

Redis là một hệ quản lý cơ sở dữ liệu NoSQL (Not Only SQL) dựa trên cấu trúc dữ liệu key-value. Nó được sử dụng phổ biến để lưu trữ dữ liệu tạm thời, cache, và truy vấn nhanh các khóa dữ liệu trong ứng dụng.

Spring Data Redis là một module của Spring Framework được sử dụng để làm việc với cơ sở dữ liệu Redis. Spring Data Redis cung cấp các tính năng và tiện ích để làm việc với Redis trong một ứng dụng Spring

4. Configuration trong Java

Thêm dependency vào pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

```
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Config redis cache in properties file:

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

Thêm annotation @EnableCaching vào Spring Boot main class để kích hoạt việc sử dụng cache trong ứng dụng Spring. Khi đánh dấu một class cấu hình hoặc một bean configuration với @EnableCaching, Spring sẽ tìm kiếm và cấu hình một cache manager cho ứng dụng. Sau đó, các annotation như @Cacheable, @CachePut và @CacheEvict có thể được sử dụng để quản lý cache cho các phương thức của các bean trong ứng dụng.

Sử dụng Jedis — a simple and powerful Redis client implementation

```
@Bean
JedisConnectionFactory jedisConnectionFactory() {
    JedisConnectionFactory jedisConFactory
      = new JedisConnectionFactory();
    jedisConFactory.setHostName("localhost");
    jedisConFactory.setPort(6379);
    return jedisConFactory;
}
@Bean
public RedisTemplate<String, Object> redisTemplate() {
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(jedisConnectionFactory());
    return template;
}
```

Define RedisTemplate sử dụng jedisConnectionFactory. This can be used for querying data with a custom repository

Trong framework Redisson của Spring Boot, @RedisHash("Student") được sử dụng để đánh dấu một lớp dữ liệu (class) là một Redis Hash, được lưu trữ trong Redis.

Dùng @RedisHash("Student") giúp Redisson tự động ánh xạ các đối tượng của lớp Student vào Redis và tiện lợi cho việc lưu trữ và truy xuất dữ liệu trong ứng dụng Spring Boot của bạn

Redis là một hệ thống cơ sở dữ liệu key-value và không hỗ trợ truy vấn dựa trên SQL như các cơ sở dữ liệu quan hệ truyền thống. Vì vậy, khái niệm của JpaRepository trong Spring Data JPA không áp dụng trực tiếp cho Redis

Tham khảo:

https://howtodoinjava.com/spring-boot/spring-boot-cache-example/

https://geeksbox.net/caching-rest-service-spring-boot-voi-redis/

https://viblo.asia/p/huong-dan-spring-boot-redis-aWj53NPGl6m

https://kungfutech.edu.vn/bai-viet/spring-boot/su-dung-redis-trong-spring-boot

https://docs.spring.io/spring-data/redis/docs/3.1.6/reference/html/#redis

https://redis.io/docs/get-started/document-database/

https://www.docker.com/blog/how-to-use-the-redis-docker-official-image/

https://www.atlantic.net/dedicated-server-hosting/how-to-install-redis-server-using-docker-container/

https://stackoverflow.com/questions/41371402/connecting-to-redis-running-in-docker-container-from-host-machine

5. @TimeToLive

@RedisHash(timeToLive = 3600)	@TimeToLive
@RedisHash(timeToLive = 3600) *public class MyObject {* *@Id* *private String id;* *private String data;* *// Constructors, getters, setters, etc.* *}*	*@RedisHash("myObject")* *public class MyObject {* *@Id* *private String id;* *private String data;* *@TimeToLive* *private Long expiration;*

	// Constructors, getters, setters, etc. *}*
Là 1 annotation trong Spring Data Redis, được sử dụng để đánh dấu một lớp đối tượng là một Redis hash key và cấu hình thời gian sống (TTL) cho toàn bộ hash key đó. Annotation này chỉ ra rằng các đối tượng thuộc lớp đó sẽ tồn tại trong Redis trong một khoảng thời gian xác định và sau đó sẽ bị xóa. Trong ví dụ trên, cho biết rằng các đối tượng của lớp này sẽ tồn tại trong Redis trong 3600 giây (1 giờ) và sau đó sẽ bị xóa.	Là 1 annotation trong Spring Data Redis, được sử dụng để đánh dấu một trường dữ liệu trong một đối tượng Redis là trường thời gian sống (TTL). Đây là một cách để quản lý thời gian sống của dữ liệu tại mức trường dữ liệu trong Redis. Trong ví dụ trên, trường expiration được đánh dấu bởi @TimeToLive, cho biết rằng giá trị của trường này là thời gian sống của đối tượng trong Redis. Khi thời gian sống hết hạn (expiration), đối tượng sẽ tự động bị xóa khỏi Redis. Giá trị của @TimeToLive ghi đè lên giá trị được đặt bởi @RedisHash(timeToLive) nếu cả hai đều được sử dụng trong cùng một lớp.

6. @Indexed

@Indexed là một annotation trong Spring Data, được sử dụng để đánh dấu một thuộc tính trong lớp là một trường có khả năng tìm kiếm. Khi bạn đánh dấu một thuộc tính bằng @Indexed, nó sẽ được tạo thành một chỉ mục trong cơ sở dữ liệu, giúp tìm kiếm dữ liệu hiệu quả hơn.

```
@Data
@RedisHash(value = "user", timeToLive = 60L)
@JsonInclude(JsonInclude.Include.NON_NULL)
public class User implements Serializable {
    @Id
    private String id;
    @Indexed // Đánh dấu thuộc tính name là một trường có khả năng tìm kiếm
    private String name;
    private String salary;
}
```

Trong ví dụ trên, thuộc tính name của lớp User sẽ được tạo thành một chỉ mục, cho phép tìm kiếm dữ liệu dựa trên giá trị của trường này. Khi bạn thực hiện tìm kiếm theo tên, Redis sẽ sử dụng chỉ mục đã tạo để tối ưu hóa việc truy vấn dữ liệu.

7.

IV. Spring Security

SERVLET APPLICATIONS

As most open source projects, Spring Security deploys its dependencies as Maven artifacts.

❖ Spring Boot with Maven

Spring Boot provides a *spring-boot-starter-security* starter that aggregates Spring Security-related dependencies

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

Since Spring Boot provides a Maven BOM to manage dependency versions, you do not need to specify a version. If you wish to override the Spring Security version, you can do so by providing a Maven property:

```
<properties>
  <spring-security.version>6.3.0</spring-security.version>
</properties>
```

Spring Security integrates with the Servlet Container by using a standard Servlet Filter. This means it works with any application that runs in a Servlet Container. More concretely, you do not need to use Spring in your Servlet-based application to take advantage of Spring Security.

Spring Security, 1 phần của Spring Framework, là framework hỗ trợ lập trình viên triển khai các biện pháp bảo mật ở cấp độ ứng dụng. Ở cấp độ cơ sở hạ tầng, chúng ta có thể áp dụng một số phương pháp, ví dụ như tường lửa (firewall), proxy server, whitelist IP, ... Tuy nhiên, vẫn cần phải xây dựng bảo mật ở cấp độ ứng dụng, đặc biệt với những ứng dụng có logic phân quyền người dùng phức tạp.

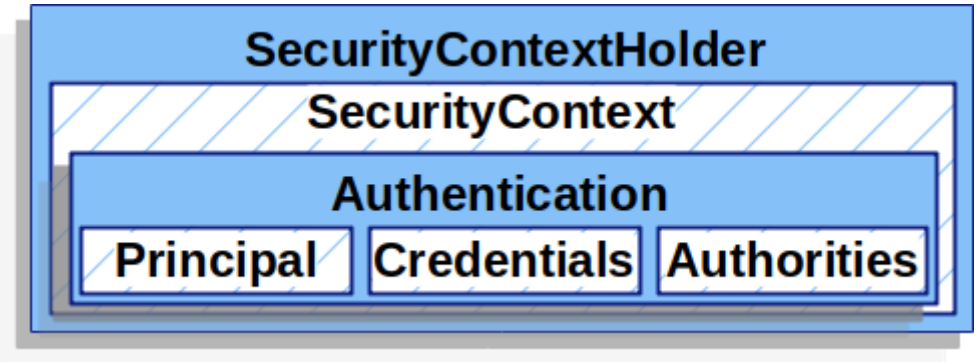
Spring Security hoạt động xoay quanh 2 vấn đề chính là authentication và authorization ở cấp độ Web request cũng như cấp độ method invocation.

1. Tổng quan cơ chế hoạt động

Kiến trúc của Spring Security trong các ứng dụng web hoàn toàn dựa trên các Servlet Filter, nó không phụ thuộc sử dụng servlet, không phụ thuộc vào bất kì một công nghệ web nào.

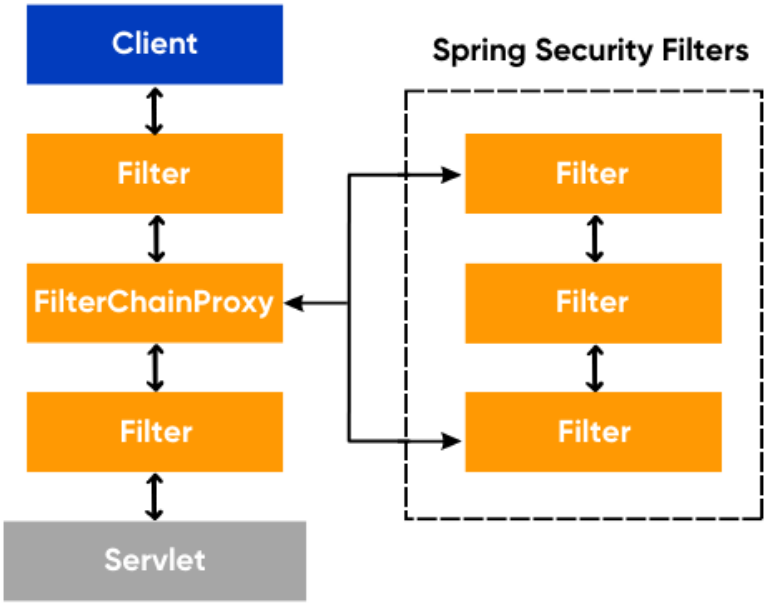
Về cơ bản, Spring Security gồm nhiều filter khác nhau, được quản lý thông qua FilterChainProxy. Các filter này có trách nhiệm chặn request, kiểm tra xác thực, kiểm tra uỷ quyền và redirect to Controller để xử lý request. Throw exception khi thông tin xác thực không hợp lệ.

Spring Security cung cấp class SecurityContext, quản lý tất cả thông tin xác thực của ứng dụng. Thông tin xác thực của người dùng sẽ được lưu trong đối tượng Authentication, và được lấy thông qua SecurityContextHolder. SecurityContext sử dụng ThreadLocal để quản lý thông tin => đảm bảo các phương thức thực thi trên cùng một luồng sẽ được chia sẻ thông tin xác thực như nhau.



- SecurityContextHolder là nơi mà Spring Security lưu trữ thông tin của người dùng được xác thực.
- SecurityContext được lấy từ SecurityContextHolder và chứa Authentication của người dùng được xác thực hiện tại.
- Authentication có thể là input cho AuthenticationManager để cung cấp thông tin đăng nhập (credentials) mà user đã cung cấp để xác thực hoặc user hiện tại từ SecurityContext.
- GrantedAuthority – Một quyền được cấp cho principal trên (ví dụ: roles, scopes, ...)
- AuthenticationManager - API định nghĩa cách mà filters của SpringSecurity thực hiện việc xác thực.
- ProviderManager - the most common implementation of AuthenticationManager.
- AuthenticationProvider – được sử dụng bởi ProviderManager để thực hiện 1 loại xác thực cụ thể
- Request Credentials with AuthenticationEntryPoint – sử dụng cho việc yêu cầu thông tin đnăg nhập từ client (i.e. redirecting to a log in page, sending a WWW-Authenticate response, etc.)
- AbstractAuthenticationProcessingFilter – một Filter cơ sở được sử dụng cho việc xác thực. Đây là 1 idea cho luồng cấp cao của việc xác thực và cách chúng hoạt động cùng nhau.

2. Architecture



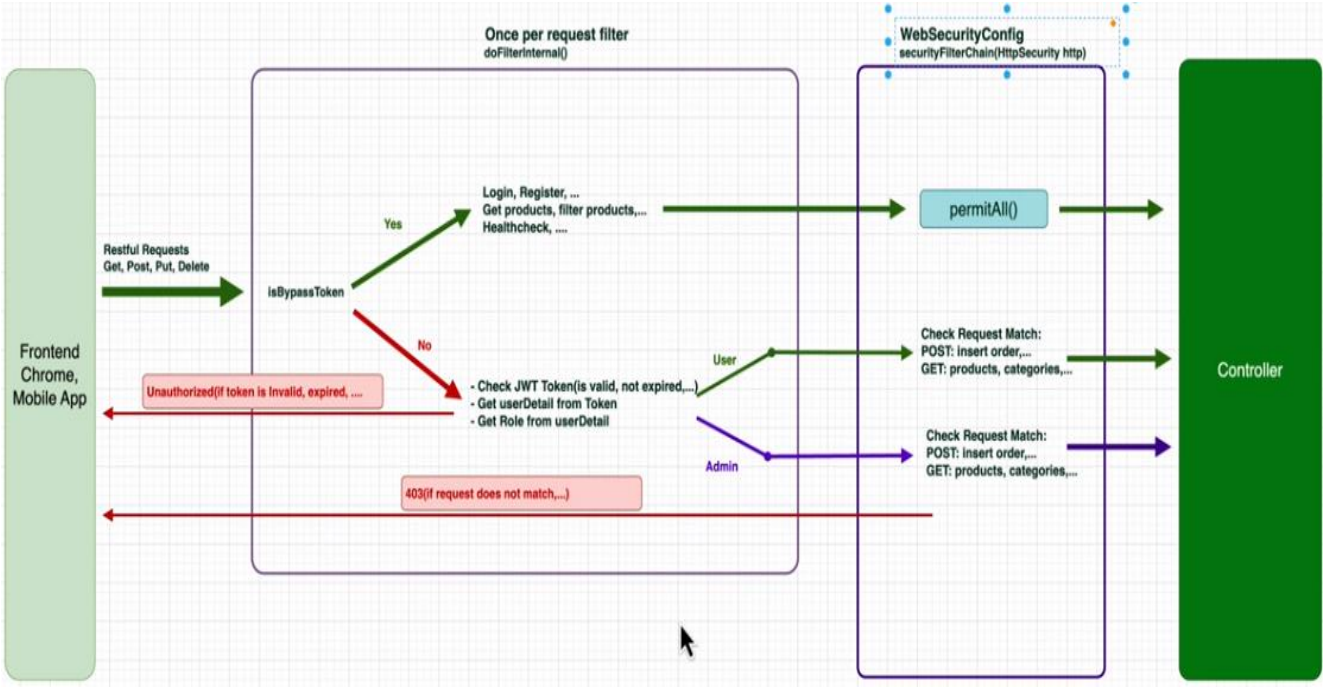

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
   // do something before the rest of the application
    chain.doFilter(request, response); // invoke the rest of the application
    // do something after the rest of the application
}
```

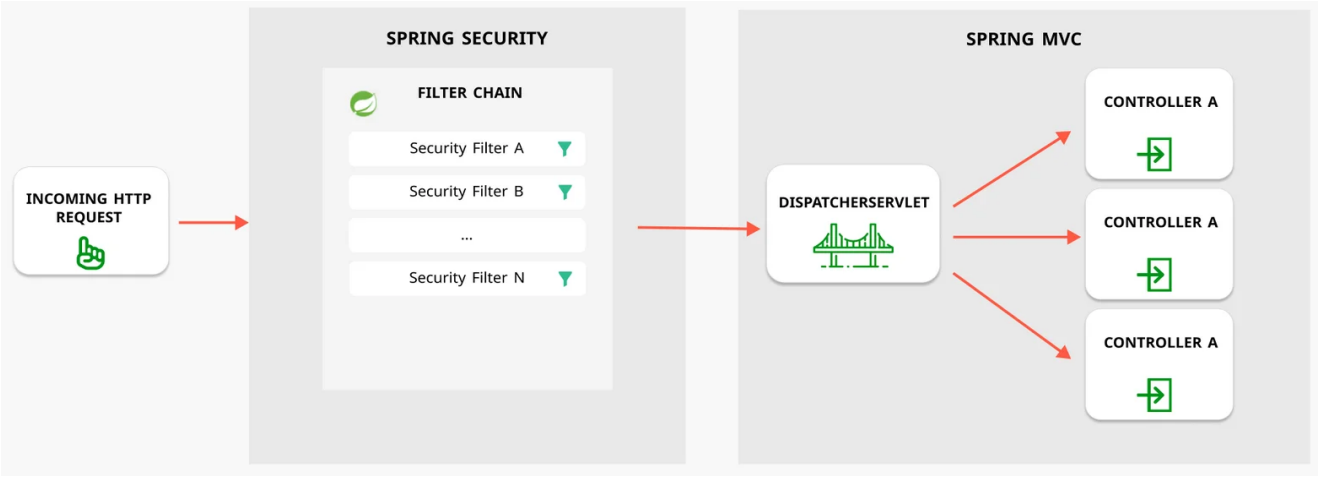
Filter là 1 đối tượng để chặn các HTTP request và response (trước khi gửi request tới controller), với mục đích chuyển user request tới 1 trang khác, hoặc chặn truy cập vào 1 trang web nếu user không có quyền hoặc ghi lại các thông tin log

Khi 1 request gửi đến application, request phải đi qua các Filters => đến đích. Mỗi filter sẽ có nhiệm vụ cụ thể, như kiểm tra xác thực, phân quyền, điều hướng hoặc xử lý các lỗi bảo mật. Nếu tại 1 filter, doFilter() không được gọi thì request sẽ không đến được đích và dừng lại tại filter đó. Ngược lại, nếu request được chấp nhận bởi Spring Security, nó sẽ được tiếp tục xử lý bởi web application như bình thường. Khi web application return a response for a request, nó cũng sẽ được chuyển qua lại chuỗi các filters của Spring Security để áp dụng các thiết lập bảo mật cho response.

Tập các filter được nối với nhau => FilterChain

Servlet Filter là các filter được sử dụng trong lập trình Servlet





Một filter có 3 method cơ bản:

- public void doFilter (ServletRequest, ServletResponse, FilterChain): thực hiện các tác vụ xử lý và chuyển tiếp request đến Filter tiếp theo hoặc Servlet
- public void init(FilterConfig filterConfig): thực hiện các công việc khởi tạo cần thiết cho Filter trước khi nó được sử dụng để xử lý các request
- public void destroy(): thực hiện các công việc dọn dẹp hoặc giải phóng tài nguyên mà Filter có thể đã sử dụng trong quá trình xử lý các request

Một số loại filter phổ biến:

- Authentication Filter: Sử dụng để xác thực người dùng dựa trên thông tin đăng nhập

Ví dụ: UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, JwtAuthenticationFilter, OAuth2AuthenticationFilter, RememberMeAuthenticationFilter, …

- Authorization Filter: Sử dụng để kiểm tra quyền truy cập của user

Ví dụ: FilterSecurityInterceptor, RoleAuthorizationFilter, IPAuthorizationFilter, AnonymousAuthenticationFilter, ExceptionTranslationFilter, …

- Logging Filter: Ghi lại thông tin request và response, theo dõi hoạt động và kiểm tra lỗi
- Compression Filter (Filter nén): Nén dữ liệu trước khi gửi response cho client
- Encoding Filter (Filter mã hóa): Đảm bảo rằng input and ouput data được mã hóa đúng cách, giúp tránh các lỗ hổng bảo mật như Cross-Site Scripting (XSS).
- Transaction Filter (Filter giao dịch): Sử dụng để quản lý giao dịch, đảm bảo tính toàn vẹn của dữ liệu và quyền rollback trong trường hợp có lỗi xảy ra.
- Caching Filter (Filter bộ nhớ đệm): Lưu trữ bản sao của các response đã tạo => giảm số lần thực hiện.
- CORS Filter (Filter quản lý chính sách CORS): Quản lý chính sách Cross-Origin Resource Sharing (CORS), cho phép hoặc từ chối các yêu cầu từ các nguồn khác nhau.

Spring Security is a framework that provides authentication, authorization, and protection against common attacks. Là 1 tiêu chuẩn bảo mật của 1 Spring application

Spring Security tích hợp với Servlet Container bằng cách sử dụng Servlet Filter tiêu chuẩn. Spring Security setup 1 Servlet Filter tên là FilterChainProxy để thực hiện bảo mật và xác thực. Một số các filter trong FilterChainProxy như:

- BasicAuthenticationFilter: Là 1 loại Authentication Filter, để xác thực người dùng bằng phương thức Basic Authentication. Username&password được mã hóa dưới dạng Base64 nhằm đảm bảo an toàn khi truyền qua mạng, password vẫn được mã hóa bằng BCryptPasswordEncoder khi lưu vào db => Dễ bị tấn công Man-in-the-middle
- UsernamePasswordAuthenticationFilter: Là 1 loại Authentication Filter, để xác thực người dùng qua phương thức Username-Password Authentication.
- FilterSecurityInterceptor: Là 1 loại Authorization Filter, để kiểm soát quyền truy cập đối với các request

FilterSecurityInterceptor hoạt động sau quá trình xác thực. Sau khi người dùng đã xác thực thành công và có phiên làm việc (session), FilterSecurityInterceptor sẽ kiểm tra quyền truy cập của user đối với các tài nguyên hoặc chức năng trong ứng dụng.

Cách hoạt động của FilterSecurityInterceptor như sau:

- Nó nhận request từ user sau khi đã xác thực thành công.
- Kiểm tra thông tin về quyền truy cập và dựa vào đó FilterSecurityInterceptor sẽ cho phép hoặc từ chối truy cập.
- Nếu truy cập được phép, request tiếp tục được xử lý bởi các Filter và Servlet tiếp theo.

3. Authentication

Trước khi truy cập các tài nguyên giới hạn quyền của hệ thống, user phải đăng nhập. Spring Security có cơ chế xử lý đăng nhập cho các bài toán từ đơn giản tới phức tạp, ví dụ:

- Thông tin đăng nhập được quản lý bởi chính hệ thống
- Thông tin đăng nhập được quản lý bởi bên thứ 3
- Cơ chế xác thực SSO (Oauth2, SAML)
- Cơ chế xác thực LDAP

Spring Security cung cấp interface AuthenticationProvider để xử lý các bài toán nêu trên một cách đơn giản, hiệu quả, an toàn và dễ mở rộng. Authentication Provider là một thành phần quan trọng trong Spring Security chịu trách nhiệm xác minh thông tin xác thực của người dùng hoặc ứng dụng. Ví dụ, khi một người dùng đăng nhập vào hệ thống, Authentication Provider sẽ kiểm tra thông tin đăng nhập của người dùng và trả về kết quả xác thực.

Authentication Provider được sử dụng bởi Authentication Manager để xử lý yêu cầu xác thực từ người dùng hoặc ứng dụng. Mỗi Authentication Provider chỉ hỗ trợ một loại Authentication cụ thể như: UsernamePasswordAuthenticationToken, JwtAuthenticationToken, PreAuthenticatedAuthenticationToken, …

Spring Security hỗ trợ xác thực thông qua một số cơ chế, bao gồm:

- Form-based authentication: Xác thực thông qua một form đăng nhập.
- HTTP Basic authentication: Xác thực thông qua các header authorization.
- Authentication via a custom login page: Xác thực thông qua một trang đăng nhập tùy chỉnh.

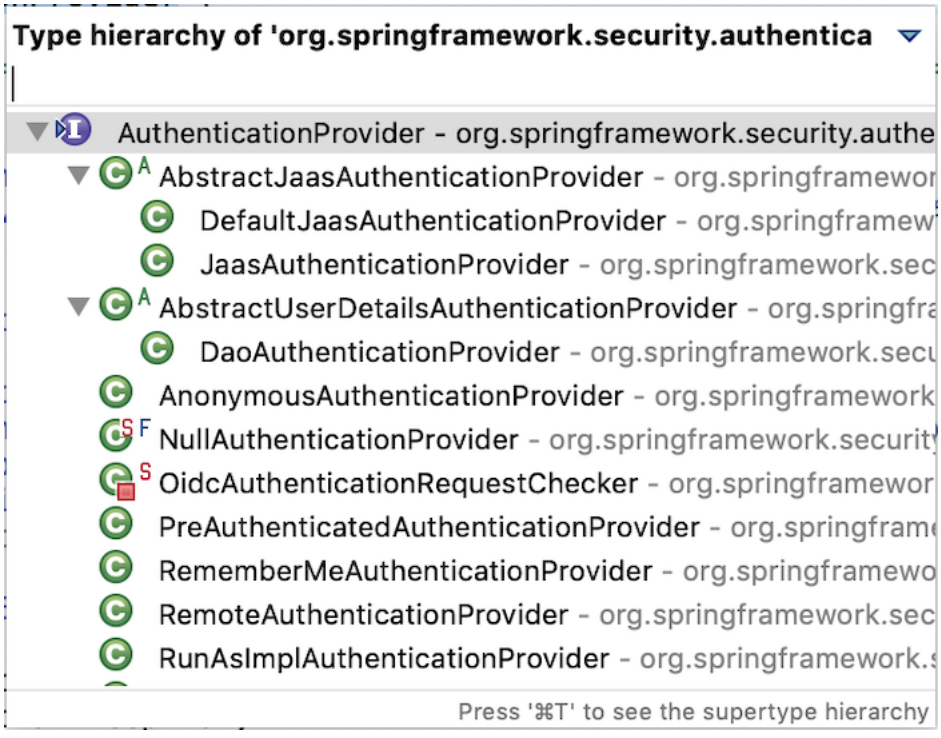
- Pre-authenticated authentication: Xác thực thông qua các giá trị được cung cấp từ phía client

Quy trình authentication

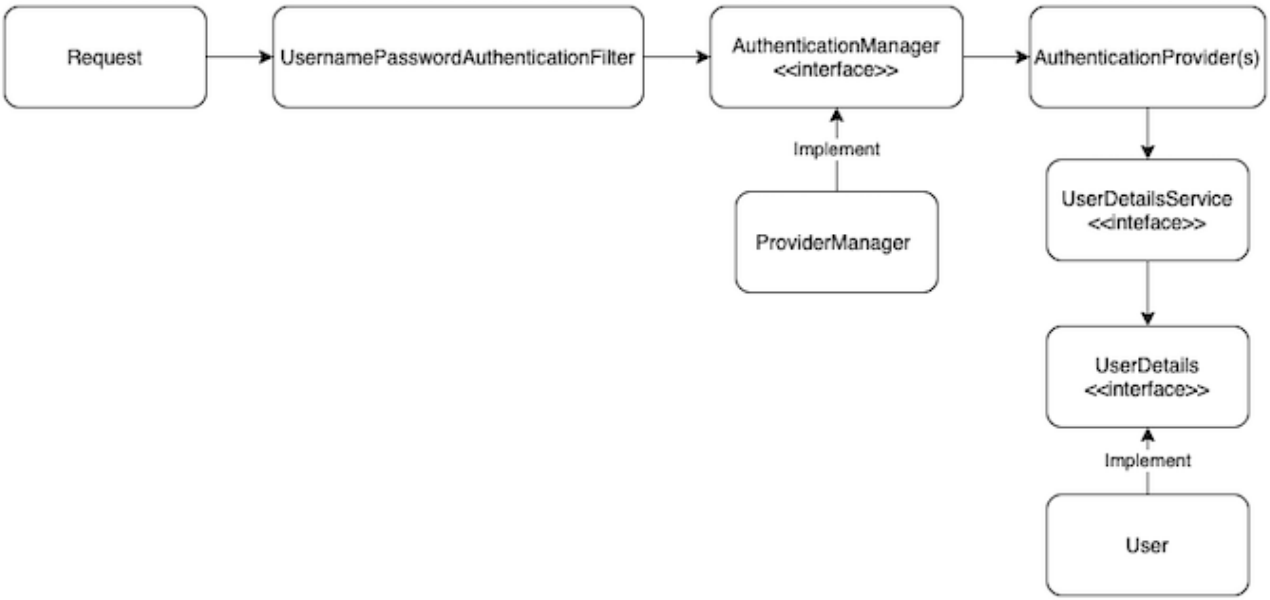
Khi gọi tới request, request phải đi qua các Filters được gọi bởi class FilterChainProxy để Spring Security có thể thực hiện trách nhiệm của mình. Mỗi filter có chức năng và nhiệm vụ riêng.

Ví dụ: class UsernamePasswordAuthenticationFilter và class BasicAuthenticationFilter sẽ đảm nhận cho việc xử lý authentication trong Spring Security. UsernamePasswordAuthenticationFilter sẽ đảm nhận việc authentication khi user gọi đến POST request "/login" còn BasicAuthenticationFilter sẽ đảm nhận việc authentication với thông tin user và password được truyền trên header của request. Cả 2 filter này đều gọi đến phương thức authenticate() của interface AuthenticationManager để thực hiện việc authentication.

Nói thêm về interface AuthenticationManager thì implementation của interface thường dùng nhất là class ProviderManager. Class ProviderManager sẽ gọi một list các AuthenticationProvider mà chúng ta khai báo trong tập tin cấu hình của Spring Security, security.xml

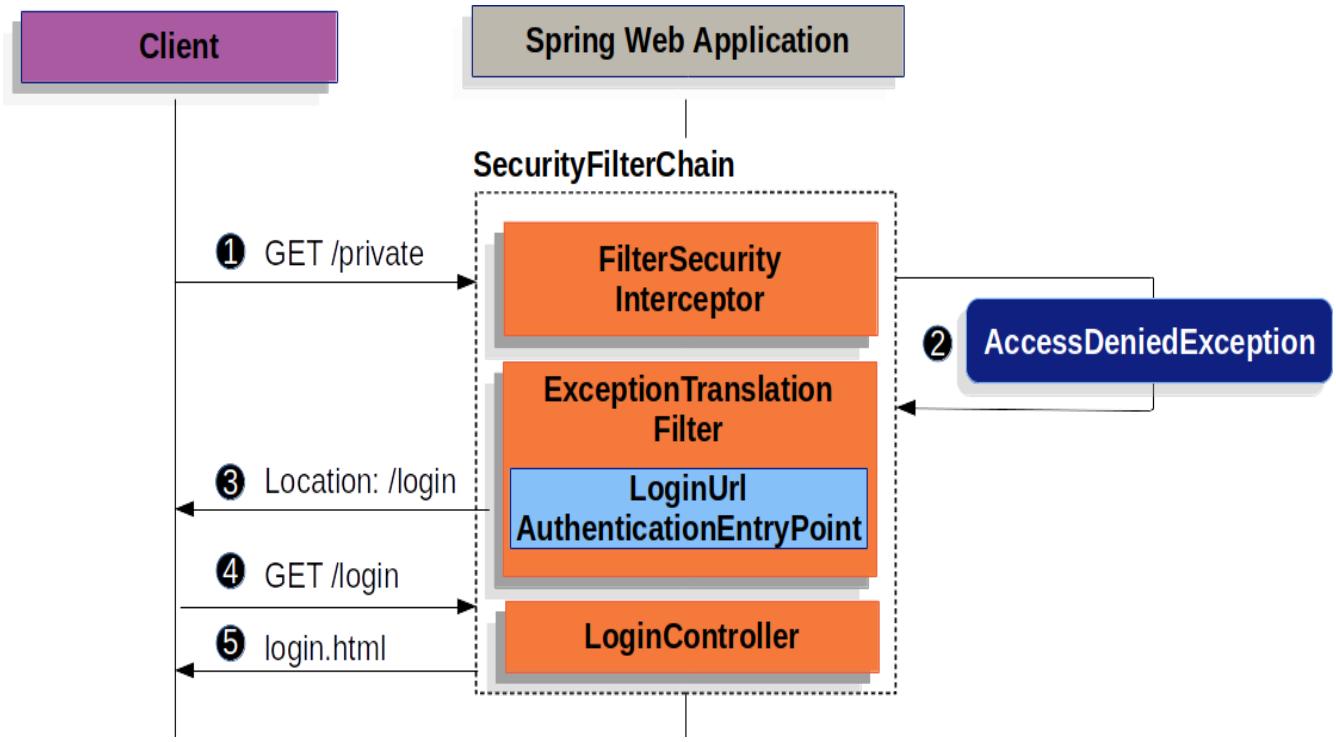


Mặc định thì class DaoAuthenticationProvider sẽ đảm nhận việc authentication. Nó sẽ sử dụng class UserDetailsService để lấy thông tin user mà chúng ta khai báo trong tập tin cấu hình security.xml để làm việc này.



❖ **Reading Username and Password**

UsernamePasswordAuthenticationFilter



1/ User gửi 1 unauthenticated request đến resource mà chưa được cấp quyền

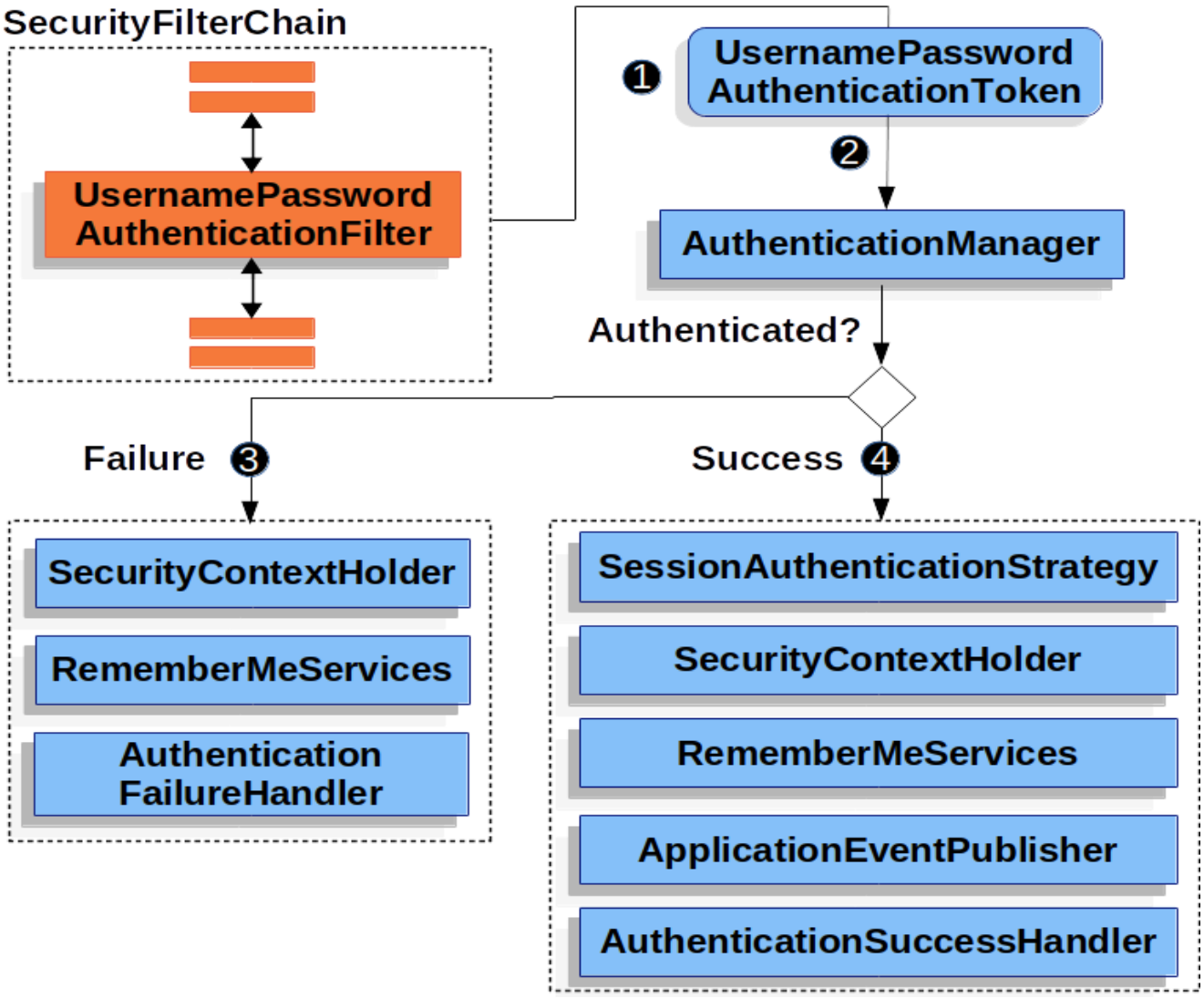
2/ AuthorizationFilter kiểm tra request và request bị từ chối bằng cách throw AccessDeniedException

3/ Vì user chưa xác thực, ExceptionTranslationFilter thực hiện xác thực và chuyển hướng sang login page với AuthenticationEntryPoint đã định. Thường AuthenticationEntryPoint là 1 instance của LoginUrlAuthenticationEntryPoint

4/ Yêu cầu login page

5/ Trả về login page

Khi POST username&password, UsernamePasswordAuthenticationFilter thực hiện xác thực.



1/ UsernamePasswordAuthenticationFilter tạo ra 1 UsernamePasswordAuthenticationToken (1 loại Authentication), bằng cách trích xuất username&password từ HttpServletRequest instance

2/ UsernamePasswordAuthenticationToken được chuyển cho AuthenticationManager instance để xác thực.

3/ Nếu xác thực thất bại:

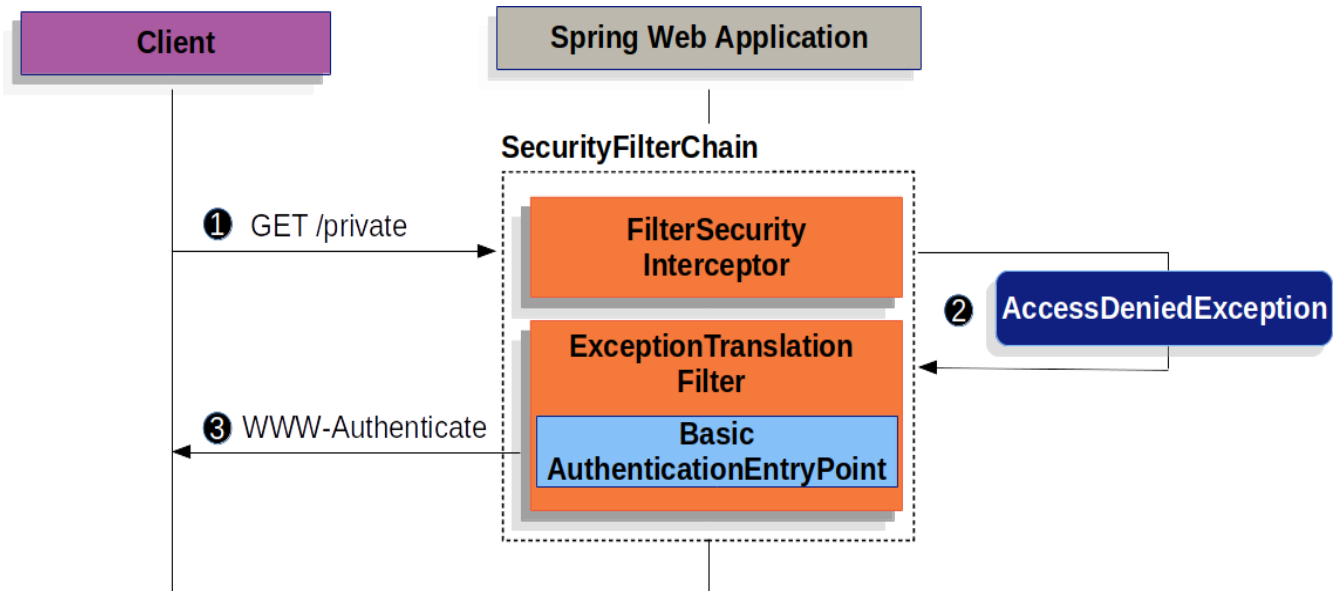
- The SecurityContextHolder bị xóa (SecurityContextHolder lưu trữ thông tin về Principal (user hiện tại) và các roles liên quan đến user đó)
- RememberMeServices được gọi, nếu nó không được config => không hoạt động (RememberMeServices hỗ trợ tính năng "Remember Me" để giữ cho người dùng đăng nhập tự động sau khi hết thời gian phiên hoặc sau khi đóng trình duyệt)
- AuthenticationFailureHandler được gọi để xử lý xác thực thất bại

4/ Nếu xác thực thành công:

- SessionAuthenticationStrategy được thông báo về 1 lần new login (SessionAuthenticationStrategy sử dụng để quản lý session sau khi người dùng login)
- Thông tin user hiện tại được đặt trong SecurityContextHolder.
- RememberMeServices được gọi
- ApplicationEventPublisher xuất ra InteractiveAuthenticationSuccessEvent (1 sự kiện trong Spring Security, sử dụng để thông báo về sự kiện xác thực thành công)
- AuthenticationSuccessHandler được gọi sau khi login thành công. AuthenticationSuccessHandler cho phép xử lý các hành động cụ thể sau khi login thành công. Cụ thể, SimpleUrlAuthenticationSuccessHandler sẽ chuyển hướng user đến một URL sau khi login thành công.

Spring security có 1 default login page luôn được enable.

BasicAuthenticationFilter

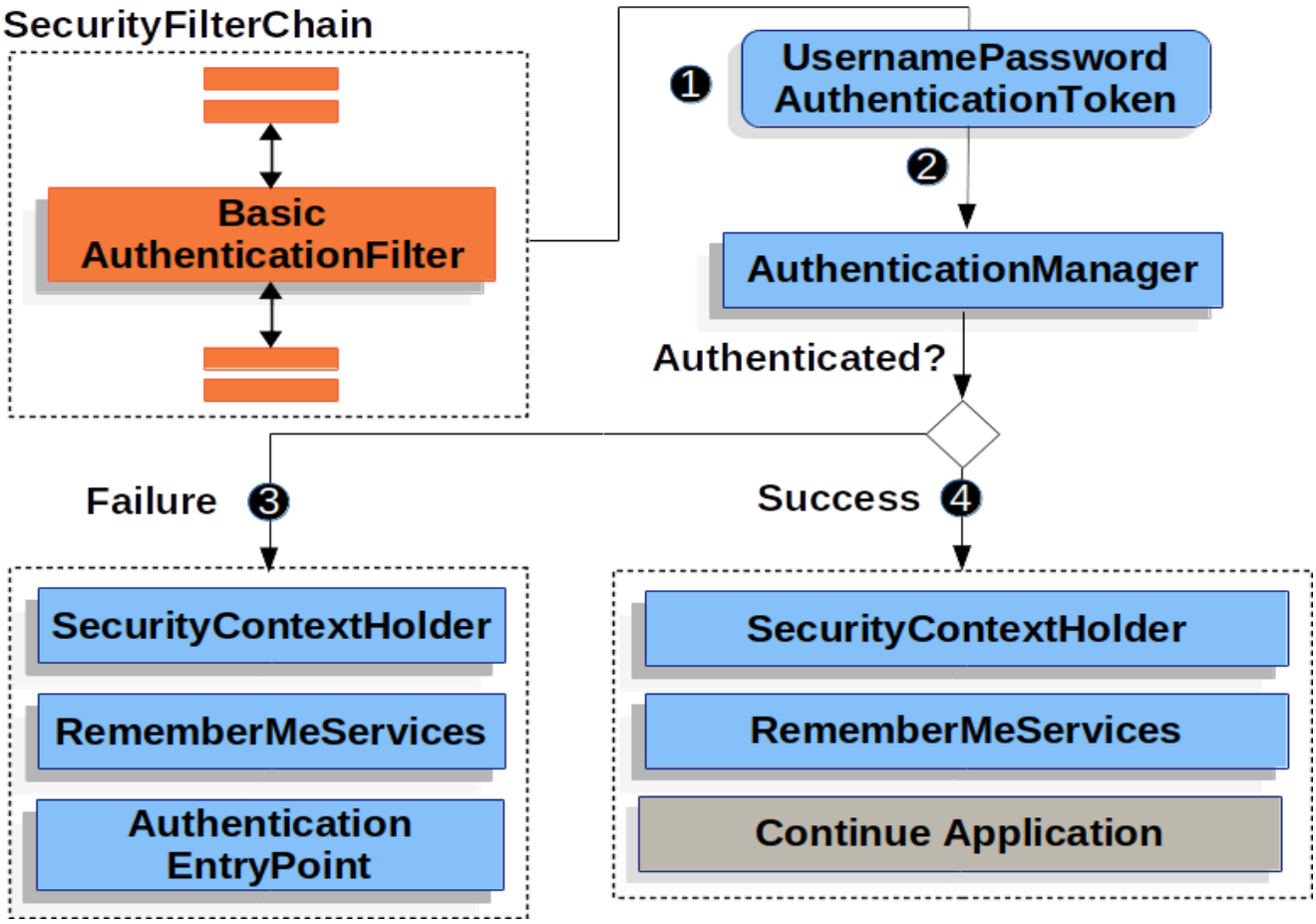


1/ User gửi 1 unauthenticated request đến resource mà chưa được cấp quyền

2/ AuthorizationFilter kiểm tra request và request bị từ chối bằng cách throw AccessDeniedException

3/ Vì user chưa được xác thực, ExceptionTranslationFilter thực hiện xác thực. AuthenticationEntryPoint là 1 instance của BasicAuthenticationEntryPoint, sẽ gửi 1 WWW-Authenticate header cho client. RequestCache thường là NullRequestCache (không lưu lại request) vì client có thể gửi lại request. (RequestCache: lưu trữ thông tin request trước đó. Sau khi user login hoặc authen thành công, người dùng sẽ được chuyển hướng trở lại request gốc)

Khi client nhận được WWW-Authenticate header, thì phải nhập lại username&password



1/ BasicAuthenticationFilter tạo ra 1 UsernamePasswordAuthenticationToken (1 loại Authentication) bằng cách trích xuất username&password từ HttpServletRequest instance

2/ UsernamePasswordAuthenticationToken được chuyển cho AuthenticationManager instance để xác thực.

3/ / Nếu xác thực thất bại:

- The SecurityContextHolder bị xóa
- RememberMeServices được gọi, nếu nó không được config => không hoạt động
- AuthenticationEntryPoint được gọi để gửi lại 1 WWW-Authenticate header cho client

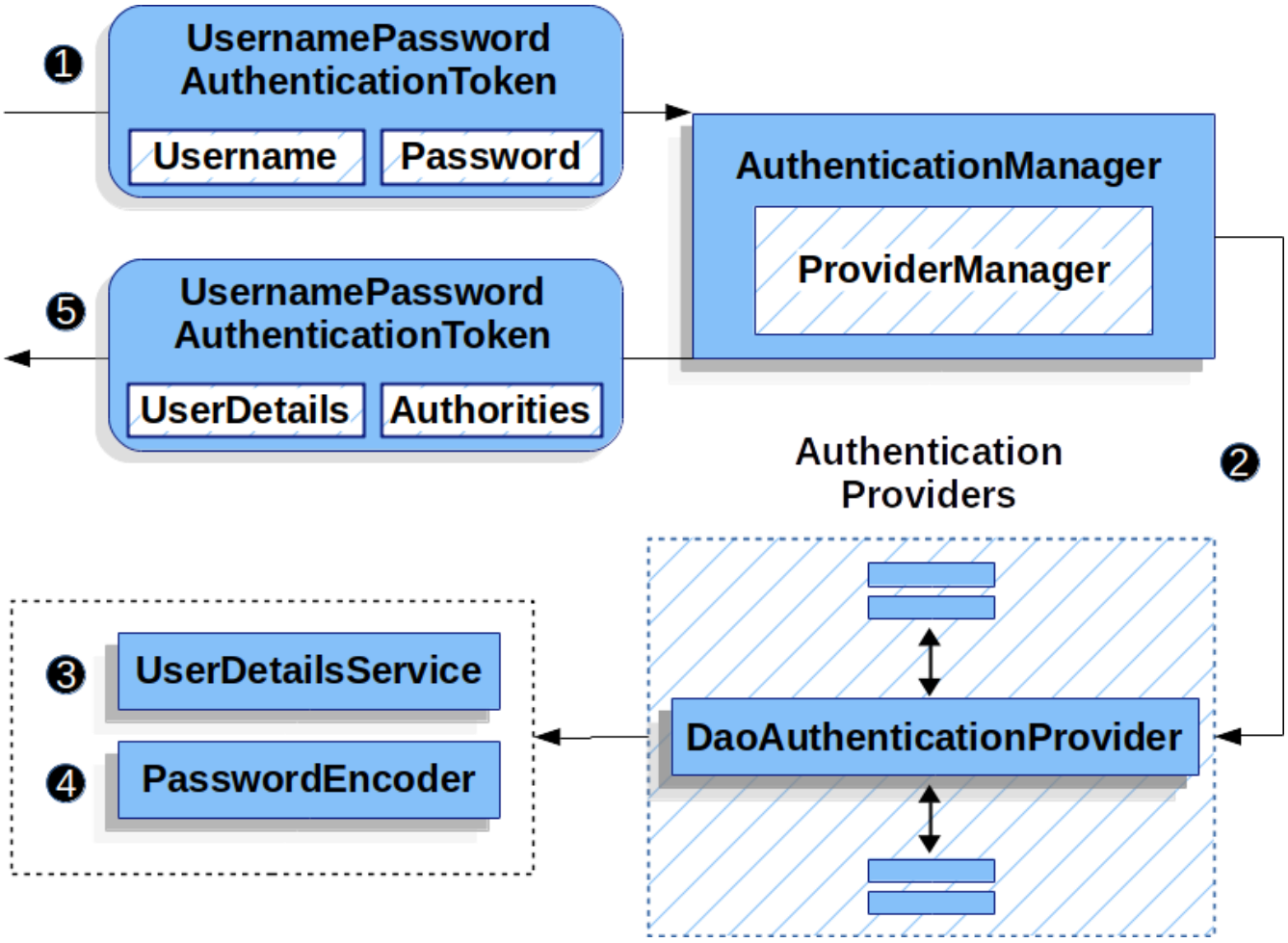
4/ Nếu xác thực thành công:

- Thông tin người dùng đặt trong SecurityContextHolder.
- RememberMeServices được gọi
- BasicAuthenticationFilter gọi doFilter(request,response) để tiếp tục các phần còn lại.

Spring security có HTTP Basic Authentication là luôn được enable

- **DaoAuthenticationProvider**

DaoAuthenticationProvider là 1 AuthenticationProvider implementation rằng sử dụng UserDetailsService and PasswordEncoder để xác thực username and password.



1/ The authentication Filter from the Reading the Username & Password section gửi 1 UsernamePasswordAuthenticationToken đến AuthenticationManager, cái được implements bởi ProviderManager.

2/ The ProviderManager is configured to use an AuthenticationProvider of type DaoAuthenticationProvider.

3/ DaoAuthenticationProvider looks up the UserDetails from the UserDetailsService.

4/ DaoAuthenticationProvider uses the PasswordEncoder to validate the password on the UserDetails returned in the previous step.

5/ When authentication is successful, the Authentication that is returned is of type UsernamePasswordAuthenticationToken and has a principal that is the UserDetails returned by the configured UserDetailsService. Ultimately, the returned UsernamePasswordAuthenticationToken is set on the SecurityContextHolder by the authentication Filter.

❖ LDAP

4. Authorization

Spring Security hỗ trợ rất tốt trong việc phân quyền ứng dụng ở 2 cấp độ: request level và method invocation level.

Spring Security cung cấp interface GrantedAuthority để đại diện cho các quyền của người dùng. Nếu một hệ thống phức tạp, cần kiểm tra ủy quyền dựa trên nhiều yếu tố khác, Spring Security cũng cung cấp thêm đối tượng User để lập trình viên có thể tùy chỉnh theo ý muốn.

Spring Security cung cấp tính năng ủy quyền ở cấp độ request bằng việc kế thừa class WebSecurityConfigurerAdapter. Ví dụ, các url khác nhau sẽ có quyền hạn khác nhau.

Tuy nhiên, đối với một hệ thống phức tạp, việc kiểm soát truy cập đến tận cấp độ phương thức (method) là điều cần thiết. Giống như bảo mật ở cấp độ request trong ứng dụng web nêu trên, Spring cũng cung cấp cả các tính năng bảo mật ở cấp độ Class và Method. Điều đặc biệt là việc này rất dễ dàng thực hiện bằng cách sử dụng các annotation của Spring Security: @PreAuthorize, @PostAuthorize, @Secured

Phạm vi ảnh hưởng của annotation phụ thuộc vào vị trí mà nó được đánh dấu. Nếu đánh dấu các annotation ở đầu class, nó sẽ áp dụng cho tất cả các phương thức của class đó. Trong trường hợp sử dụng riêng lẻ, nó chỉ ảnh hưởng với phương thức được đánh dấu. Đối với các phương thức bên trong annotation, lập trình viên có thể sử dụng các phương thức có sẵn của Spring Security, cũng như có thể tùy chỉnh theo các bài toán cụ thể.

Ngoài các tính năng chính nêu trên, Spring Security còn hỗ trợ nhiều tính năng khác như:

- Hỗ trợ nhiều password encoder
- Dễ dàng xử lý các exceptions
- Cấu hình CSRF, CORS đơn giản
- Tương thích dễ dàng với các template engine của Java như Thymeleaf

@PreAuthorize	@PostAuthorize
The @PreAuthorize can check for authorization before entering into method. The @PreAuthorize authorizes on the basis of role or the argument which is passed to the method.	The @PostAuthorize checks for authrorisation after method execution. The @PostAuthorize authorizes on the basis of logged in roles, return object by method and passed argument to the method. For the returned object spring security provides built-in keyword i.e. returnObject.

@PreAuthorize	@PostAuthorize	@Secured

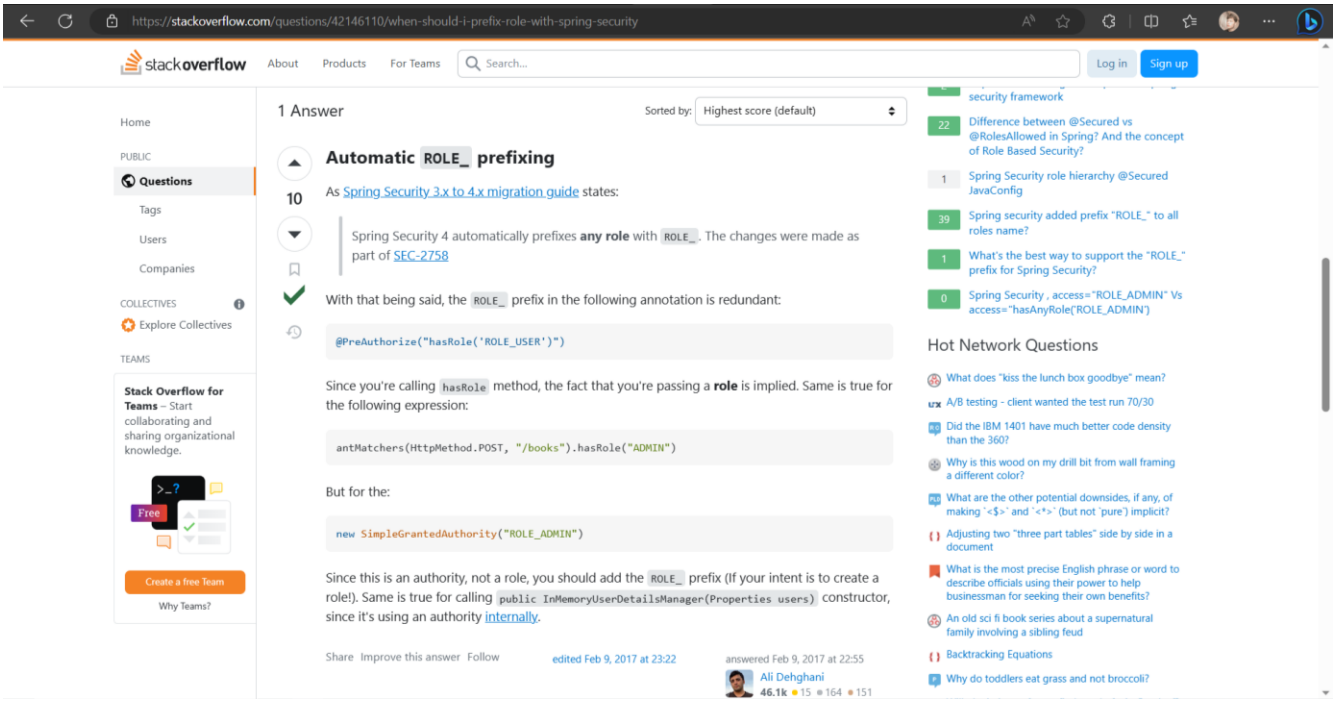
Spring Security allows you to model your authorization at the request level. For example, with Spring Security you can say that all pages under /admin require one authority while all other pages simply require authentication.

By default, Spring Security requires that every request be authenticated. That said, any time you use an HttpSecurity instance, it's necessary to declare your authorization rules.

Role/Authority

Các vai trò (Roles) hoặc quyền (Authorities) là nhóm các quyền hoặc chức năng mà người dùng có thể được cấp trong ứng dụng.

- hasRole: kiểm tra xem user có một vai trò cụ thể hay không. Ví dụ: ADMIN, USER
- hasAuthority: kiểm tra xem user có một quyền cụ thể hay không. Ví dụ: READ, WRITE



hasRole	hasAuthority
hasRole("ADMIN")	*hasAuthority("ADMIN")*
@PreAuthorize("hasRole('ROLE_ADMIN')")	*@PreAuthorize("hasAuthority ('ADMIN')")*
new SimpleGrantedAuthority("ROLE_ADMIN")	*new SimpleGrantedAuthority("ADMIN")*

Cũng có thể dùng @PreAuthorize("isAuthenticated()") để cho mọi người dùng có thể truy cập

NOTE:

Mặc định, khi tạo các đối tượng SimpleGrantedAuthority và thêm vào danh sách authorities, Spring Security sẽ tự động thêm tiền tố "ROLE_" vào tên vai trò. Điều này giúp Spring Security nhận biết rằng các đối tượng này đại diện cho vai trò trong hệ thống.

Tuy nhiên, khi bạn gọi getAuthorities(), Spring Security sẽ loại bỏ tiền tố "ROLE_" khỏi tên vai trò trước khi trả về danh sách các GrantedAuthority. Điều này làm cho danh sách trở nên dễ đọc và sử dụng hơn.

Vì vậy, bạn không thấy tiền tố "ROLE_" khi sử dụng getAuthorities(), nhưng khi bạn sử dụng các annotation như @Secured hoặc @PreAuthorize, bạn vẫn cần sử dụng tiền tố "ROLE_" trong tên vai trò để xác định quyền truy cập.

Mặc định, quy tắc ủy quyền dựa trên vai trò bao gồm "ROLE_" as a prefix. Nghĩa là nếu có authorization rule yêu cầu security context có vai trò là "USER", Spring Security mặc định sẽ tìm

GrantedAuthority#getAuthority trả về "ROLE_USER". You can customize this with GrantedAuthorityDefaults. GrantedAuthorityDefaults exists to allow customizing the prefix to use for role-based authorization rules:

```
@Bean
static GrantedAuthorityDefaults grantedAuthorityDefaults() {
  return new GrantedAuthorityDefaults("MYPREFIX_");
}
```

5. Stateful and stateless

HTTP (Hypertext Transfer Protocol) là một giao thức không lưu trạng thái (stateless protocol). Điều này có nghĩa là mỗi request (yêu cầu) HTTP độc lập và không có sự nhớ trạng thái từ các request trước đó. Mỗi request chứa đủ thông tin để máy chủ (server) hiểu và xử lý nó mà không cần phải biết về bất kỳ thông tin liên quan đến các request trước đó.

Stateful Authentication

After successful authentication, the application generates a random token to send back to the client and then creates a client authenticated session in memory or an internal database. When a client tries to access the application with a given token, the application tries to retrieve session data from session storage, checks if the session valid, and then decides whether the client has access to the desired resource or not.

Stateless Authentication

After successful authentication, the application generates a token with all necessary data, signs it with a public key, and sends it back to a client. There is a standard for token generation, it is JWT (JSON Web Token). The process is described in OpenID Connect (OIDC) specification. When a client tries to access the application with a token, the application verifies the token sign with a private key, checks if the token is expired, retrieves all session data from the token, and makes a decision if a client has access to the desired resource.

Revoke hoặc invalidate token/ logout => user không làm được => stateless không xử lý. Ngược lại, stateful thì server nắm key, có thể remove được

Cơ chế xác thực dựa trên mô hình stateless còn được gọi là xác thực dựa trên mã thông báo (token-based authentication). Ví dụ, bước đầu tiên, người dùng gửi username/password đến server để

xác thực. Nếu thông tin chính xác, server sẽ mã hóa thông tin và xử lý, sau đó trả về một mã thông báo cho client. Client sẽ lưu trữ mã thông báo, với mỗi yêu cầu sau đó, client đính kèm mã thông báo đó trên header. Server sẽ xử lý xác thực dựa trên mã thông báo đó.

Cơ chế xác thực qua mã thông báo khắc phục được những nhược điểm của cơ chế xác thực qua phiên. Giảm một lượng lớn dữ liệu cần lưu trữ ở server, phù hợp với các hệ thống sử dụng RESTful API, dễ dàng mở rộng và phân cụm. Tuy nhiên phương pháp này cũng cần xử lý phức tạp hơn.

Có một số kiểu mã thông báo phổ biến như:

- Basic token: Khi người dùng đăng nhập, server sẽ trả về cho client mã thông báo. Server sẽ lưu mã thông báo trong database hoặc cache storage. Với mỗi request gửi kèm token, server sẽ kiểm tra token đó còn hiệu lực hay không và đang được gắn với người dùng nào trong database. Với kiểu này, server vẫn phải hoạt động khá nhiều.
- Json web token (JWT): Giống như basic token, server sẽ trả về mã JWT cho client khi người dùng đăng nhập. Nhưng thay vì lưu ở cả server và client, JWT này sẽ lưu trữ ở client, và được server giải mã thông qua secret key. JWT có thể được sử dụng như một cơ chế xác thực không yêu cầu database.

Spring cung cấp thư viện JWT cũng như các thư viện mã hoá phục vụ cho cơ chế xác thực dựa trên mã thông báo. Với việc xử lý mã thông báo, để đảm bảo việc xử lý chỉ diễn ra một lần duy nhất cho từng yêu cầu, Spring cung cấp filter OncePerRequestFilter để lập trình viên có thể triển khai cơ chế trên một cách nhanh chóng và chính xác.

6. Token

JWT

https://blog.codegym.vn/2020/02/20/huong-dan-ve-spring-security-va-jwt-phan-1/

https://codegym.vn/blog/huong-dan-su-dung-spring-boot-security-voi-oauth2/

7. Annotations

❖ **@PreAuthorize(), @PostAuthorize(), @Secured()**

Muốn sử dụng @PreAuthorize(), @PostAuthorize(), @Secured() cần phải thêm @EnableGlobalMethodSecurity(prePostEnabled = true)

❖

Tham khảo:

https://huongdanjava.com/vi/spring-security

https://rabiloo.com/vi/blog/nang-cao-bao-mat-voi-spring-security

https://spring.io/projects/spring-security

https://200lab.io/blog/co-che-hoat-dong-cua-spring-security/

https://huongdanjava.com/vi/authentication-trong-spring-security.html

https://huongdanjava.com/vi/tong-quan-ve-quy-trinh-xu-ly-request-trong-spring-security.html

V. Test trong springboot

Viết Test là một phần quan trọng trong việc xây dựng ứng dụng

Sử dụng dependency: spring-boot-starter-test

Vấn đề 1: Biết rằng Spring Boot sẽ phải tạo Context và tìm kiếm các Bean. Sau tất cả các bước config và khởi tạo thì chúng ta sử dụng @Autowired để lấy đối tượng ra sử dụng. Vậy khi viết Test, làm sao @Autowired bean vào class Test được? làm sao cho JUnit hiểu @Autowired là gì?

Giải quyết => tích hợp Spring vào với JUnit bằng cách sử dụng @RunWith(SpringRunner.class)

Spring Boot đã thiết kế ra lớp SpringRunner để giúp tích hợp Spring + JUnit => trong mọi class Test, chúng ta sẽ thêm @RunWith(SpringRunner.class) lên đầu Class test đó.

```
@RunWith(SpringRunner.class)
public class TodoServiceTest {
   ...
}
```

Khi chạy, TodoServiceTest sẽ tạo ra một Context riêng để chứa bean trong đó => có thể @Autowired thoải mái trong nội hàm Class này.

Vấn đề 2: làm sao đưa Bean vào trong Context? => @SpringBootTest và @TestConfiguration

- @SpringBootTest

@SpringBootTest sẽ đi tìm kiếm class có gắn @SpringBootApplication và từ đó đi tìm toàn bộ Bean và nạp vào Context.

⇨ Chỉ nên sử dụng trong Integration Tests, vì nó sẽ tạo toàn bộ Bean, không khác gì chạy cả cái SpringApplication.run(App.class, args); gây rất tốn thời gian, nhiều Bean thừa thãi không sử dụng

- @TestConfiguration

@TestConfiguration giống với @Configuration => tự định nghĩa ra Bean.

Các Bean được tạo bởi @TestConfiguration chỉ tồn tại trong môi trường test => phù hợp với việc viết UnitTest. Class Test nào, cần Bean gì, thì tự tạo ra trong @TestConfiguration

Vấn đề 3: Chúng ta tạo ra Rest API đón request người dùng, vậy làm sao để test nó? Nếu Controller không được test => một lỗ hổng lớn, vì nó là đầu ra chính của chương trình, nó sai => hỏng. Tuy nhiên, không thể khởi động Tomcat Server + Database để test 1 API => tốn tài nguyên và thời gian.

Spring Boot hỗ trợ test Controller mà không cần khởi động Tomcat Server bằng annotation @WebMvcTest.

Tất nhiên là nếu không khởi động Server, thì phải có một phương thức khác giả lập.

Ví dụ:

```
@RunWith(SpringRunner.class)
// Bạn cần cung cấp lớp Controller cho @WebMvcTest
@WebMvcTest(TodoRestController.class)
public class TodoRestControllerTest {
   // MockMvc do Spring cung cấp, có tác dụng giả lập request, thay thế việc khởi động Server
   @Autowired
   private MockMvc mvc;
}
```

1. JUnit test

JUnit là một framework phổ biến trong việc thực hiện testing cho ứng dụng Java.

Sử dụng dependency: junit-jupiter-api

Một số lý do tại sao nên sử dụng JUnit:

- Đảm bảo tính ổn định: Giúp xác minh rằng mã nguồn hoạt động đúng và ổn định
- Tự động hóa kiểm thử: Cho phép tự động hóa quy trình kiểm thử, tiết kiệm thời gian và nguồn lực (có thể chạy các bộ kiểm thử một cách liên tục mỗi khi có thay đổi trong mã nguồn)

- Giúp bảo trì mã nguồn: Khi thay đổi mã nguồn, các bộ kiểm thử JUnit giúp xác minh rằng thay đổi không làm ảnh hưởng đến các tính năng khác của ứng dụng.

Ví dụ:

```
public class MyMathTest {
   @Test
   public void testAddition() {
      MyMath math = new MyMath();
      int result = math.add(2, 3);
      Assertions.assertEquals(5, result); //so sánh kết quả thực tế với giá trị mong đợi
   }
}
```

Quy tắc đặt tên test case: Tên của test case nên bắt đầu bằng "test", theo sau là mô tả về điều kiện hoặc hành vi muốn kiểm tra. Ví dụ: *testAdditionWithPositiveNumbers*

JUnit hoạt động bằng cách tìm kiếm và chạy các test case đã viết một cách tự động.

Lifecycle của một test case

- Setup (Khởi tạo): JUnit tạo một instance mới của test case trước khi chạy mỗi test case. Điều này đảm bảo rằng mỗi test case hoạt động độc lập với các test case khác.
- Chạy test case: JUnit chạy các phương thức được gắn thẻ @Test trong test case.
- Kiểm tra kết quả: Sau khi chạy một test case, JUnit kiểm tra kết quả và xác định xem test case đó đã thành công hay thất bại.
- Cleanup (Dọn dẹp): JUnit loại bỏ instance của test case sau khi đã hoàn thành, giải phóng tài nguyên và chuẩn bị cho test case tiếp theo.

JUnit cung cấp một loạt các phương thức kiểm tra kết quả của test case:

- assertEquals(expected, actual): So sánh giá trị actual với giá trị expected. Nếu chúng không giống nhau, test case sẽ thất bại.
- assertTrue(condition): Kiểm tra xem condition có đúng hay không. Nếu condition là true, test case sẽ thành công
- assertFalse(condition): Kiểm tra xem condition có sai hay không. Nếu condition là false, test case sẽ thành công.

- assertNotNull(object): Kiểm tra xem object có khác null hay không. Nếu object khác null, test case sẽ thành công.
- ...

❖ **Annotations**

@Test: Test case là một lớp Java đơn giản được gắn thẻ bằng @Test để chỉ ra rằng nó là một bộ kiểm thử

@MockBean: Spring hỗ trợ mock với @MockBean, chúng ta có thể lấy ra một Bean "giả" mà không thèm để ý tới thằng Bean "thật" (dù thằng "thật" có ở trong Context đi nữa, cũng không quan tâm).

❖

2. Mockito

Tư tưởng của Mock đơn giản là thay vì tạo ra 1 đối tượng thật sự, ta chỉ tạo ra 1 đối tượng giả mạo, có đầy đủ chức năng như thật (nhưng không phải thật)

Sử dụng dependency: mockito-core

❖ **Enable Mockito annotations**

- Sử dụng MockitoJUnitRunner

```
@ExtendWith(MockitoExtension.class)
public class MockitoAnnotationUnitTest {...}
```

- MockitoAnnotations.openMocks()

```
@BeforeEach
public void init() {   MockitoAnnotations.openMocks(this);}
```

- MockitoJUnit.rule() (make our rule public)

```
public class MockitoAnnotationsInitWithMockitoJUnitRuleUnitTest {
   @Rule
   public MockitoRule initRule = MockitoJUnit.rule();
}
```

❖ **@Mock annotation**

- @Mock is used to create and inject mocked instances without having to call Mockito.mock manually

Using @Mock	Calling Mockito.mock without using @Mock
@Mock *List<String> mockedList;*	*List mockList = Mockito.mock(ArrayList.class);*

```
public class MockingExample {
    @Test
    public void testMockBehavior() {
        // Khởi tạo đối tượng giả lập
        MyService mockService = Mockito.mock(MyService.class);

        // Thiết lập hành vi cho mock object
        Mockito.when(mockService.doSomething()).thenReturn("Hello, Mockito!");

        // Kiểm tra hành vi
        String result = mockService.doSomething();
        // check kết quả xem có đúng không
        assertEquals("Hello, Mockito!", result);
        // check xem mockService.doSomething() đã được gọi hay chưa
        Mockito.verify(mockService, times(1)).doSomething();
    }
```

❖ **@Spy annotation**

- @Spy is used to spy on an existing instance (@Spy được sử dụng để tạo 1 object giả, giả lập một số phương thức cụ thể của nó. Điều này hữu ích khi muốn giữ nguyên một phần thực thi thật của đối tượng và có thể sử dụng các method Mockito như when, thenReturn để định nghĩa hành vi của object giả trong quá trình kiểm tra.

Using @Spy	Without using @Spy
@Spy *List<String> spiedList = new ArrayList<String>();*	*List<String> spyList = Mockito.spy(new ArrayList<String>());*

```
public class SpyExample {
    private List<String> list = Mockito.spy(new ArrayList<String>());
```

```
@Test
public void testSpy() {
    list.add("one");

    // Spy vẫn ghi lại các phương thức thực thi thật
    assertEquals(1, list.size());

    // Spy cho phép bạn thay đổi hành vi cho các phương thức giả lập
    Mockito.when(list.size()).thenReturn(100);

    assertEquals(100, list.size());
}
```

❖ **@Captor annotation**

- @Captor annotation creates an ArgumentCaptor instance. ArgumentCaptor giúp kiểm tra các đối số được truyền vào cho các method trong các lời gọi mock.

```
public class CaptorExample {
    @Mock
    private List<Object> list;

    @Captor
    private ArgumentCaptor<Object> captor;

    @Test
    public void testCaptor() {
        list.add(1);

        // Ghi lại tham số của phương thức add
        Mockito.verify(list).add(captor.capture());

        // Kiểm tra tham số đã được ghi lại
        assertEquals(1, captor.getValue());
    }
```

❖ **@InjectMock annotation**

- @InjectMocks annotation is used to inject mock fields into the tested object (inject 1 đối tượng giả lập vào một đối tượng thực tế để kiểm tra hành vi của nó)

❖ **Injecting a Mock into a Spy**

- If we want to use a mock with a spy, we can manually inject the mock through a constructor:

```
@Mock
Map<String, String> wordMap;
@Spy
MyDictionary spyDic = new MyDictionary();
MyDictionary(Map<String, String> wordMap) {  this.wordMap = wordMap;}
```

- Instead of using the annotation, we can now create the spy manually:

```
@Mock
Map<String, String> wordMap;
MyDictionary spyDic;
@BeforeEach
public void init() {
   MockitoAnnotations.openMocks(this);
   spyDic = Mockito.spy(new MyDictionary(wordMap));}
```

❖ **BDDMockito**

BDD giúp cho việc viết test tự nhiên và dễ đọc hiểu. BDD defines a structure of writing test, gồm 3 phần: given some preconditions, when an action occurs, then verify the ouput.

- Traditional mockito: Sử dụng when(object).then() và verify()
- BDDMockito provides BDD aliases for various Mockito methods: given().will() và then()

❖ **Mockito's Mock method**

❖

VI.

C/ Other Annotations

1. **@Controller**

@Controller và @RestController là hai annotation trong Spring Framework. @Controller được sử dụng để đánh dấu một lớp là một Controller trong mô hình MVC của Spring, và thường được sử dụng để xử lý các yêu cầu HTTP và trả về các view hoặc model.

Trong khi đó, @RestController được sử dụng để xác định một Controller làm việc với dữ liệu RESTful, và mỗi phương thức của nó trả về dữ liệu thô thay vì view

2. **@RestController**

Cơ bản giống @Controller

3. **@RequestMapping**

Example:

@RequestMapping(path="/design", produces="application/json")

"Produces": Any of handler methods will only handle requests if the request's "Accept" header includes "application/json"

4. **@CrossOrigin**

5. **@NoArgsConstructor**

@NoArgsConstructor là một nnotation dùng để định nghĩa một constructor (hàm khởi tạo) không có tham số cho một lớp (class).

@NoArgsConstructor được sử dụng để tạo ra một constructor không có tham số cho lớp đó.

Một số tham số:

- Access: được sử dụng để chỉ định mức truy cập của constructor được tạo ra
  Ví dụ: *@NoArgsConstructor(access = AccessLevel.PRIVATE)* chỉ định rằng constructor sẽ có mức truy cập là 'private', nghĩa là chỉ có thể truy cập từ bên trong cùng lớp đó. Điều này có nghĩa là constructor không thể được gọi từ các lớp bên ngoài hoặc kế thừa từ lớp đó. Thường thì việc tạo ra một constructor riêng tư không tham số như vậy sẽ được sử dụng trong các trường hợp mà

người lập trình muốn giới hạn việc khởi tạo đối tượng từ bên ngoài và buộc các phương thức tạo đối tượng khác phải được sử dụng

-

.

6. @Converter

@Converter là một annotation trong Java Persistence API (JPA) được sử dụng để đánh dấu một lớp chuyển đổi (converter). Chuyển đổi là một cơ chế trong JPA để chuyển đổi giữa kiểu dữ liệu của một thuộc tính trong đối tượng và kiểu dữ liệu tương ứng trong cơ sở dữ liệu.

Để sử dụng @Converter, bạn cần tạo một class chuyển đổi riêng biệt và áp dụng chú thích @Converter lên class đó. Class chuyển đổi phải implements interface *javax.persistence.AttributeConverter* và cung cấp phương thức để chuyển đổi giữa kiểu dữ liệu của đối tượng và kiểu dữ liệu trong cơ sở dữ liệu.

Ví dụ:

```
@Converter
public class MyConverter implements AttributeConverter<MyEnum, String> {
    @Override
    public String convertToDatabaseColumn(MyEnum attribute) {
        // Chuyển đổi từ MyEnum sang String để lưu vào cơ sở dữ liệu
    }

    @Override
    public MyEnum convertToEntityAttribute(String dbData) {
        // Chuyển đổi từ String trong cơ sở dữ liệu sang MyEnum khi đọc dữ liệu
    }
}
```

Trong ví dụ trên, chúng ta đã tạo một class MyConverter để chuyển đổi giữa kiểu dữ liệu MyEnum và kiểu dữ liệu String. Bằng cách áp dụng @Converter lên class này, JPA sẽ nhận biết và sử dụng chuyển đổi này khi lưu trữ đối tượng trong cơ sở dữ liệu.

7. @Enumerated(EnumType.STRING)

@Enumerated(EnumType.STRING) là một annotation trong Java Persistence API (JPA) được sử dụng để ánh xạ một thuộc tính enum vào một cột trong cơ sở dữ liệu.

Khi bạn đánh dấu một thuộc tính enum với @Enumerated(EnumType.STRING), JPA sẽ lưu trữ giá trị của enum dưới dạng chuỗi (string) vào cột tương ứng trong cơ sở dữ liệu thay vì lưu trữ giá trị số nguyên mặc định. Điều này giúp làm cho dữ liệu trong cơ sở dữ liệu dễ đọc và dễ hiểu hơn.

Ví dụ, giả sử bạn có một thuộc tính enum có tên "Status" với các giá trị là "ACTIVE", "INACTIVE" và "PENDING". Nếu bạn đánh dấu thuộc tính này với @Enumerated(EnumType.STRING), khi lưu trữ vào cơ sở dữ liệu, các giá trị sẽ được biểu diễn dưới dạng chuỗi "ACTIVE", "INACTIVE" và "PENDING" thay vì lưu trữ dưới dạng số nguyên như 0, 1, 2.

Với @Enumerated(EnumType.STRING), khi bạn truy xuất dữ liệu từ cơ sở dữ liệu, JPA sẽ chuyển đổi giá trị chuỗi trong cột thành giá trị enum tương ứng.

Lưu ý rằng @Enumerated(EnumType.STRING) chỉ áp dụng cho thuộc tính enum và không áp dụng cho các kiểu dữ liệu khác.

8. @JsonProperty

Annotation @JsonProperty được sử dụng để chỉ định tên của một thuộc tính trong lớp Java khi nó được chuyển đổi thành JSON hoặc khi JSON được chuyển đổi thành một đối tượng Java. Bằng cách sử dụng @JsonProperty, bạn có thể chỉ định tên khác cho một thuộc tính so với tên mặc định được sử dụng trong quá trình chuyển đổi. Điều này hữu ích khi tên thuộc tính trong Java không trùng khớp với tên trường trong JSON hoặc ngược lại.

```
public class User {
@JsonProperty("username")
private String name;
}
```

Trong ví dụ trên, khi chuyển đổi đối tượng User thành JSON, thuộc tính name sẽ được chuyển thành trường "username" trong JSON kết quả.

9. @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)

Annotation @JsonNaming được sử dụng để chỉ định chiến lược đặt tên cho các thuộc tính trong lớp Java khi chúng được chuyển đổi thành JSON. Ta thường sử dụng

PropertyNamingStrategies.SnakeCaseStrategy.class để áp dụng chiến lược đặt tên Snake Case, tức là chuyển đổi tên thuộc tính thành dạng snake_case (viết thường và phân cách bằng dấu gạch dưới).

```
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
public class User {
    private String firstName;
    private String lastName;

    // getters and setters
}
```

Trong ví dụ trên, khi chuyển đổi đối tượng User thành JSON, các thuộc tính firstName và lastName sẽ được chuyển thành các trường "first_name" và "last_name" trong JSON kết quả.

10. @CreatedBy, @LastModifiedBy và @CreatedDate, @LastModifiedDate

@CreatedBy và @LastModifiedBy được sử dụng để lưu thông tin về người dùng đã tạo hoặc chỉnh sửa một entity trong ứng dụng Spring JPA

@CreatedDate và @LastModifiedDate dùng để ghi lại thời gian tạo và thời gian chỉnh sửa entity tương ứng trong ứng dụng Spring JPA

Cách sử dụng: Đánh dấu các trường cần tạo hoặc cập nhật tự động bằng các annotation này trong class entity của bạn, sau đó Spring JPA sẽ tự động cập nhật thông tin vào các trường đó khi một thực thể được tạo mới hoặc cập nhật.

11. @Order

Trong Spring Framework, chú thích @Order được sử dụng để xác định thứ tự ưu tiên của các thành phần trong ứng dụng khi có nhiều bean hoặc các thành phần cần được sử dụng theo một thứ tự cụ thể.

Khi có nhiều bean có cùng loại hoặc kiểu, việc sắp xếp thứ tự ưu tiên có thể quyết định bean nào sẽ được ưu tiên sử dụng. Thông thường, các bean hoặc thành phần được sắp xếp theo giá trị số thứ tự trong chú thích @Order, ví dụ @Order(1), @Order(2), …

Khi Spring cần sử dụng các bean có cùng loại, nó sẽ ưu tiên sử dụng bean có độ ưu tiên cao hơn.

Chú thích @Order có thể được sử dụng trong nhiều trường hợp, chẳng hạn như xác định thứ tự của các Interceptor, các Aspect trong Aspect Oriented Programming (AOP), hoặc các thành phần khác trong ứng dụng Spring. Điều này giúp kiểm soát thứ tự thực thi hoặc ưu tiên của các thành phần trong quá trình chạy ứng dụng

12.

@JoinColumn(name = "oauth_id", foreignKey = @ForeignKey(name = "fk_oauth_client"))

13. @PrimaryKeyJoinColumn

14. @EqualsAndHashCode.Exclude

Trong Java, @EqualsAndHashCode.Exclude là một annotation (chú thích) được sử dụng trong lập trình hướng đối tượng và liên quan đến việc tạo ra các phương thức equals() và hashCode() tự động.

Trong Java, phương thức equals() được sử dụng để so sánh hai đối tượng có tương đương hay không, trong khi phương thức hashCode() được sử dụng trong việc lưu trữ đối tượng trong các cấu trúc dữ liệu dựa trên băm như HashMap hoặc HashSet.

Khi một lớp được chú thích bằng @EqualsAndHashCode.Exclude trên một trường dữ liệu, thư viện Lombok sẽ tự động bỏ qua trường dữ liệu này khi tạo ra các phương thức equals() và hashCode(). Có nghĩa rằng trường dữ liệu không được sử dụng để xác định tính tương đương của đối tượng trong quá trình so sánh và tính toán mã băm.

Việc sử dụng @EqualsAndHashCode.Exclude có thể hữu ích trong một số tình huống khi bạn muốn loại bỏ một số trường dữ liệu không quan trọng, hoặc có thể thay đổi khỏi việc so sánh tính tương đương hoặc tính toán mã băm

```
import lombok.EqualsAndHashCode;
@EqualsAndHashCode
public class Person {
    private String name;

    @EqualsAndHashCode.Exclude
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

15. @ToString.Exclude

Annotation này được sử dụng để loại trừ một trường hoặc một phương thức khỏi việc tạo mã cho phương thức toString(). Trong việc tạo chuỗi biểu diễn đối tượng thành một chuỗi, trường hoặc phương thức được đánh dấu bởi @ToString.Exclude sẽ không được bao gồm trong chuỗi kết quả.

16.

D/ NOTE

1. Arrays.asList và List.of

Arrays.asList:

- Trả về một danh sách được hỗ trợ bởi mảng được cung cấp.
- Kích thước của danh sách không thể thay đổi, nhưng các phần tử có thể được sửa đổi.
- Thích hợp cho các trường hợp bạn cần một danh sách có thể thay đổi được hỗ trợ bởi một mảng.

List.of:

- Được giới thiệu từ Java 9, tạo ra một danh sách không thể thay đổi. Cung cấp cú pháp ngắn gọn và đảm bảo tính không thể thay đổi.
- Các phần tử không thể được sửa đổi sau khi tạo, phù hợp cho các trường hợp yêu cầu tính không thể thay đổi.

Nếu cần tính đa biến và đang làm việc với các phiên bản Java trước 9, Arrays.asList là sự lựa chọn phù hợp. Tuy nhiên, nếu tính không thể thay đổi được ưu tiên hoặc sử dụng Java 9 trở lên, List.of cung cấp rõ ràng ý định và an toàn hơn trước các sửa đổi không mong muốn.

2. @Slf4j và Logger

@Slf4j: Là một annotation của project Lombok, giúp tự động tạo logger trong class khi được gắn. Dễ sử dụng và giảm việc phải viết mã boilerplate.

Logger: Là một class trong các thư viện logging như Log4j, Logback, hoặc java.util.logging. Yêu cầu khai báo và sử dụng thủ công, nhưng cung cấp linh hoạt hơn về việc tuỳ chỉnh và tính năng.

Lựa chọn giữa hai phương pháp phụ thuộc vào sở thích cá nhân, yêu cầu của dự án và sự tiện lợi. @Slf4j thích hợp cho các dự án muốn giảm mã boilerplate, trong khi Logger cung cấp tính linh hoạt cao hơn.

Trong lập trình, "boilerplate" là thuật ngữ chỉ các đoạn mã hoặc cấu trúc mã lặp đi lặp lại trong nhiều phần của một dự án hoặc giữa các dự án khác nhau. Các đoạn mã boilerplate thường là những phần cần thiết nhưng không thay đổi nhiều, chẳng hạn như khai báo biến, cấu hình, hoặc các phần giao diện người dùng cơ bản. Mã boilerplate có thể dẫn đến sự lặp lại không cần thiết và làm tăng sự phức tạp của mã nguồn. Để giảm thiểu mã boilerplate, các công cụ như các framework và thư viện được phát triển để tự động sinh mã hoặc cung cấp các cấu trúc mã sẵn có.

3. N+1 problem trong JPA

N + 1 Problem là gì?

Giả sử có 2 entity Post và PostComment có mối quan hệ many-to-one với FetchType.Lazy.

Khi ánh xạ mối quan hệ với FetchType.Lazy, các dữ liệu liên quan sẽ được tải lên khi cần thiết. Ví dụ các PostComment entity quan hệ với Post chỉ được tải lên khi gọi getPostComments().

```
List<Post> posts = em.createQuery("select p from Post p").getResultList();
for (Post post : posts) {
    System.out.println("Post Comments: " + post.getPostComments().size());
}
```

Khi thực thi đoạn code trên, Hibernate sẽ thực thi 1 câu SELECT để lấy tất cả các Post trong database. Với mỗi lần lặp qua từng Post để lấy PostComment Hibernate sẽ thực thi một câu SELECT xuống database để lấy các PostCommemt. Tổng kết lại, ta 1 câu SELECT để lấy Post và N câu SELECT để lấy các PostComment => kém hiệu quả khi có quá nhiều câu truy vấn được gửi xuống database hay còn được gọi là N + 1 Problem.

Giải pháp để tải các PostComment cùng lúc với Post là sử dụng JOIN FETCH để tránh N + 1 Problem

Nếu sử dụng FetchType.Eager rõ ràng không phải là ý tưởng hay khi dữ liệu sẽ luôn tải lên mặc cho bạn có cần sử dụng đến chúng hay không. Hơn nữa sử dụng FetchType.Eager cũng có khả năng sinh ra N + 1 Problem.

NOTE: @ManyToOne và @OneToOne mặc định là FetchType.Eager, khi bạn ánh xạ quan hệ PostComment với Post như trên, khi bạn truy vấn mà quên sử dụng FETCH JOIN khi lấy PostCommemt.

Khi sử dụng JPA – Hibermate JOIN FETCH là một trong những cách giúp tránh N + 1 hiệu quả bậc nhất, linh hoạt hơn FetchType.Eager.

https://shareprogramming.net/n-1-problem-trong-trong-jpa-va-hibernate/

https://www.vincenzoracca.com/en/blog/framework/jpa/jpa-fetch/

4. Fetch join

Mô tả ngắn gọn như này!

Trong câu truy vấn lấy user, sử dụng Fetch Join để tải thông tin của shop cùng lúc:

@Query("SELECT u FROM User u JOIN FETCH u.shop WHERE u.id = :userId")

Cách này thay cho việc sử dụng Lazy Loading cho mối quan hệ shop và Eager Loading trong định nghĩa quan hệ của User => Đảm bảo thông tin của shop sẽ luôn được tải cùng với user.

Vậy, khi nào dùng Fetch Join?

Sử dụng FETCH JOIN trong JPA là 1 cách để tải toàn bộ dữ liệu liên quan trong 1 truy vấn duy nhất, thay vì sử dụng cơ chế Lazy Loading để tải dữ liệu khi cần. Khi nào thì xem xét sử dụng FETCH JOIN:

Truy vấn hiệu suất cao: Khi cần lấy dữ liệu của nhiều đối tượng liên quan mà sử dụng Lazy Loading có thể gây ra nhiều truy vấn cơ sở dữ liệu riêng biệt => FETCH JOIN có thể cải thiện hiệu suất.

Tránh N+1 Query Problem: Khi thấy rằng sử dụng Lazy Loading dẫn đến vấn đề N+1 query (nhiều truy vấn đối với mỗi đối tượng liên quan) => FETCH JOIN giúp giảm bớt số lượng truy vấn cần thực hiện.

Truy vấn dữ liệu thường xuyên: Khi thường xuyên truy cập dữ liệu liên quan của đối tượng và muốn tránh việc tải dữ liệu bất đồng thời khi truy cập.

Truy vấn chi tiết hoặc báo cáo: Khi bạn cần lấy thông tin đầy đủ và liên quan từ nhiều đối tượng liên quan để xây dựng báo cáo hoặc hiển thị chi tiết.

Dữ liệu ít và không gây ra vấn đề hiệu suất: Khi dữ liệu ít và sử dụng FETCH JOIN không gây ra tình trạng tải dữ liệu lớn gây ra hiệu suất yếu.

Tuy nhiên, bạn cũng cần cân nhắc sử dụng FETCH JOIN một cách cẩn thận vì nó có thể gây ra vấn đề về dữ liệu trùng lặp hoặc tải quá nhiều dữ liệu một lúc, dẫn đến vấn đề hiệu suất.

ManyToOne:

Trong quan hệ ManyToOne, thông thường một đối tượng có thể tham chiếu đến nhiều đối tượng khác (ví dụ: một người có thể làm việc ở nhiều cửa hàng). Do đó, thông thường không có một "mối quan hệ 1-1" chặt chẽ giữa các đối tượng như trong quan hệ OneToOne. Vì vậy, khi bạn truy cập thông tin của một đối tượng ManyToOne, JPA thường cho phép tải thông tin từ bảng liên quan ngay lúc đó, mà không cần tải theo cơ chế Lazy Loading.

OneToOne:

Trong quan hệ OneToOne, có một sự tương ứng một-một giữa các đối tượng (ví dụ: mỗi người chỉ có một tài khoản). Do đó, cơ chế tải dữ liệu Lazy Loading thường được ưu tiên để tối ưu hiệu suất. Khi bạn truy cập thông tin của đối tượng OneToOne, JPA thường không tải thông tin liên quan ngay lúc đó mà chờ đến khi bạn thực sự cần truy cập

5.

E/