# [**Design Patterns trong Spring Framework**](https://hocspringboot.net/2021/08/22/design-patterns-trong-spring-framework/)

## Tổng quan

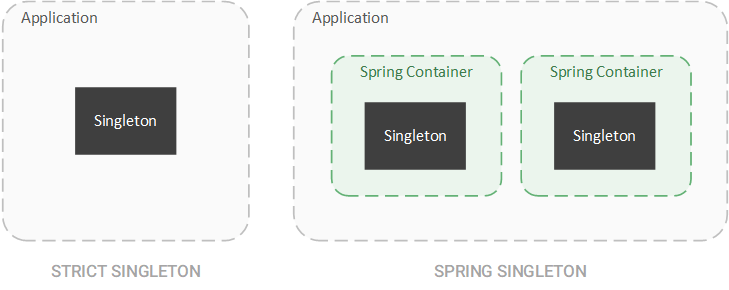
* Design Pattern là một phần thiết yếu của phát triển phần mềm, giúp giải quết các ván để lặp đi lặp lại , giúp các nhà phát triển hiểu hơn về framework bằng cách nhận ra các pattern phổ biến
* Các mẫu parttern phổ biến được sử dụng trong spring boot:
  + Singleton Pattern
  + Factory Method Pattern
  + Proxy Pattern
  + Template Pattern

## Singleton Pattern

### 2.1. Singleton Beans

- một **singleton**là duy nhất trên toàn cầu cho một ứng dụng, nhưng trong **Spring**, hạn chế này được nới lỏng. Thay vào đó, **Spring**giới hạn một singleton cho một đối tượng trên mỗi vùng chứa **Spring IoC.** Trong thực tế, điều này có nghĩa là **Spring**sẽ chỉ tạo một bean cho mỗi loại cho mỗi ngữ cảnh ứng dụng.

- Cách tiếp cận của **Spring**khác với định nghĩa chặt chẽ về **singleton**vì một ứng dụng có thể có nhiều hơn một **Spring container**



* **Mặc định thì Spring sẽ tạo tất cả các bean dưới dạng các tệp đơn.**

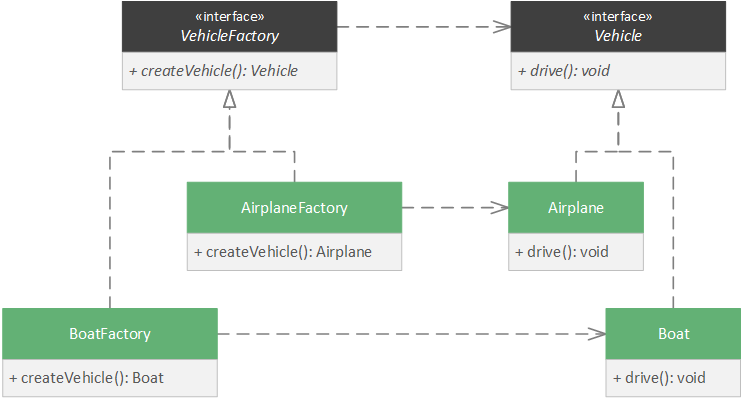
### 2.2. Autowired Singletons

- chúng ta có thể tạo hai controller trong một application context và inject một bean cùng type vào mỗi controller. Khi đó 2 đối tượng bean được inject vào trong 2 controller thực chất là giống nhau

## 3. Factory Method Pattern

- **yêu cầu một class Factory với một phương thức trừu tượng để tạo đối tượng mong muốn.**

**-** Ví dụ: ứng dụng của chúng tôi có thể yêu cầu đối tượng phương tiện. Trong môi trường hàng hải, chúng tôi muốn tạo ra tàu thuyền, nhưng trong môi trường hàng không, chúng tôi muốn tạo ra máy bay:



* chúng ta có thể tạo một factory implementation cho từng đối tượng mong muốn và trả về đối tượng mong muốn từ phương thức concrete factory.

### 3.1. Application Context

- Spring sử dụng kỹ thuật này ở gốc của Dependency Injection (DI) framwork của nó.

- Về cơ bản, Spring coi một bean container như một factory produces beans.

- Spring định nghĩa giao diện BeanFactory như một bản tóm tắt của một vùng chứa bean:

public interface BeanFactory {

getBean(Class<T> requiredType);

getBean(Class<T> requiredType, Object... args);

getBean(String name);

// ...

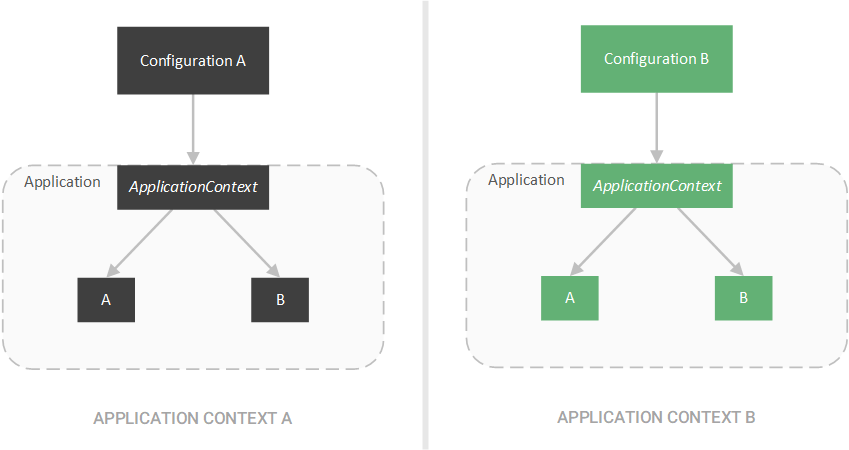
}

* Mỗi method getBean() đều được xem là factory method, trả về một bean phù hợp với các tiêu chí được cung cấp cho phương thức, như type và name của bean.
* Spring sau đó đã mở rộng BeanFactory với interface ApplicationContext, cái cấu hình ứng dụng bổ sung.
* Spring sử dụng cấu hình này để khởi động container bean dựa trên một số cấu hình bên ngoài, chẳng hạn như tệp XML hoặc annotation Java.
* Sử dụng class ApplicationContext giống như ApplicationConfigApplicationContext, sau đó chúng ta có thể tạo bean thông qua các phương thức gốc khác nhau để được kế thừa từ interface BeanFactory.

### 3.2. External Configuration

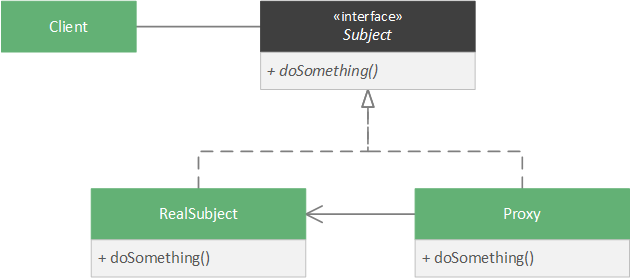
- Pattern này rất linh hoạt vì chúng ta có thể thay đổi hoàn toàn hành vi của ứng dụng dựa trên cấu hình bên ngoài.

- Nếu chúng ta muốn thay đổi việc triển khai các đối tượng tự động của ứng dụng, tôi có thể điều chình việc triển khai ApplicationContext mà tôi sử dụng.



## 4. Proxy Pattern

- **Proxy Pattern** là một kỹ thuật cho phép một đối tượng – proxy – kiểm soát quyền truy cập vào đối tượng khác – chủ thể hoặc service.



### 4.1. Transactions

Để tạo proxy, chúng tôi tạo một đối tượng triển khai giao diện giống như chủ đề của chúng tôi và chứa tham chiếu đến chủ thể.

Sau đó, chúng tôi có thể sử dụng proxy thay cho chủ thể.

Tring Spring, các bean được ủy quyền để kiểm soát quyền truy cập vào bean bên dưới. Tôi thấy cách tiếp cận này khi sử dụng các transactioin:

**@Service**

public class BookManager {

**@Autowired**

private BookRepository repository;

**@Transactional**

public Book create(String author) {

**System**.out.println(**repository**.getClass().getName());

**return** **repository**.create(**author**);

}

}

Code language: CSS (css)

Trong class BookManager, tôi đã chú thích phương thức bằng annotation @Transactional. Annotation này hướng dẫn Spring thực thi nguyên tử phương thức tạo. Nếu không có proxy, Spring sẽ không thể kiểm soát quyền truy cập vào bean BookRepository và đảm bảo tính nhất quán giao dịch của nó.

### 4.2. CGLib Proxies

Thay vào đó, Spring tạo ra một proxy bao bọc bean BookRepository và bean để thực thi phương thức tạo một cách nguyên tử.

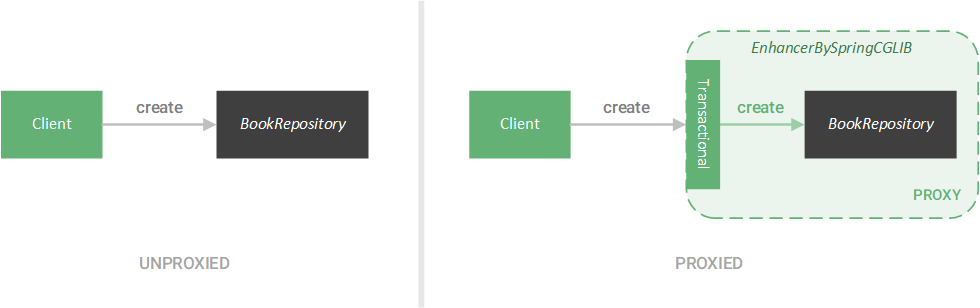
Khi chúng ta gọi phương thức tạo BookManager, có thể thấy đầu ra:

com.hdd.patterns.proxy.BookRepository$$EnhancerBySpringCGLIB$$3dc2b55c

Code language: PHP (php)

Thông thường, chúng ta mong đợi thấy một ID đối tượng BookRepository tiêu chuẩn; thay vào đó, chúng ta thấy một ID đối tượng EnhancerBySpringCGLIB.

Phía sau, Spring đã bao bọc đối tượng BookRepository của chúng ta bên trong dưới dạng đối tượng EnhancerBySpringCGLIB. Spring do đó kiểm soát quyền truy cập vào đối tượng BookRepository (đảm bảo tính nhất quán của giao dịch).



Nói tóm lại thì Spring sử dụng 2 loại proxy:

1. CGLib Proxies – được sử dụng khi proxy các lớp.
2. JDK Dynamic Proxies – được sử dụng khi proxy interface.

## 5. Template Method Pattern

Trong nhiều framework, một phần đáng kể của code là code soạn thảo.

Ví dụ: khi thực thi một câu truy vấn trên database, phải hoàn thành một loạt các bước sau:

1. Establish a connection
2. Execute query
3. Perform cleanup
4. Close the connection

Các bước này là một kịch bản lý tưởng cho **Template Method Pattern**.

### 5.1. Templates & Callbacks

Template method pattern là một kỹ thuật xác định các bước cần thiết cho một hành động, triển khai các bước soạn sẵn và để các bước có thể tùy chỉnh dưới dạng trừu tượng. Các lớp con sau đó có thể triển khai lớp trừu tượng này và cung cấp một triển khai cụ thể cho các bước còn thiếu.

Chúng ta có thể tạo một mẫu trong trường hợp truy vấn cơ sở dữ liệu của chúng tôi:

**public** **abstract** DatabaseQuery {

**public** void execute() {

Connection connection = createConnection();

executeQuery(connection);

closeConnection(connection);

}

**protected** Connection createConnection() {

// Connect to database...

}

**protected** void closeConnection(Connection connection) {

// Close connection...

}

**protected** **abstract** void executeQuery(Connection connection);

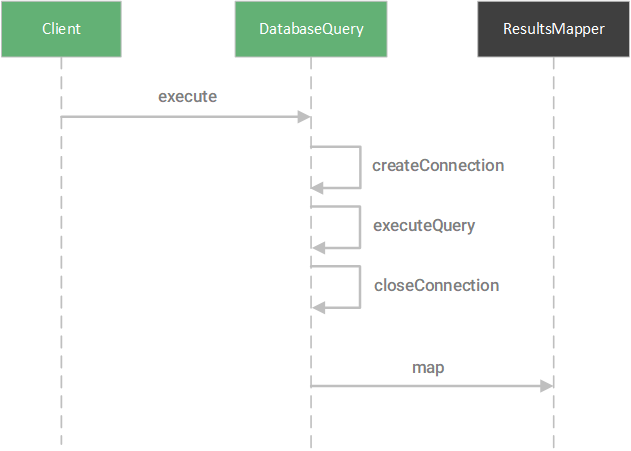
}

Code language: PHP (php)

Ngoài ra, chúng tôi có thể cung cấp bước còn thiếu bằng cách cung cấp callback method.

Callback method là một method cho phép chủ thể báo hiệu cho khách hàng rằng một số hành động mong muốn đã hoàn thành.

Trong một số trường hợp, đối tượng có thể sử dụng lệnh callback này để thực hiện các hành động – chẳng hạn như kết quả ánh xạ.



Ví dụ: thay vì có một phương thức executeQuery, chúng ta có thể cung cấp cho phương thức thực thi một chuỗi truy vấn và một phương thức gọi lại để xử lý kết quả.

Đầu tiên, chúng ta tạo phương thức gọi lại lấy một đối tượng Result và ánh xạ nó tới một đối tượng kiểu T:

**public** **interface** **ResultsMapper**<**T**> {

**public** T map(Results results);

}

Code language: PHP (php)

Sau đó, chúng tôi thay đổi lớp DatabaseQuery của mình để sử dụng lệnh gọi lại này:

**public** **abstract** DatabaseQuery {

**public** <T> T execute(String query, ResultsMapper<T> mapper) {

Connection connection = createConnection();

Results results = executeQuery(connection, query);

closeConnection(connection);

**return** mapper.map(results);

]

**protected** Results executeQuery(Connection connection, String query) {

// Perform query...

}

}

Code language: PHP (php)

Cơ chế callback này chính xác là phương pháp mà Spring sử dụng với lớp JdbcTemplate.

### 5.2. JdbcTemplate

JdbcTemplate cung cấp method truy vấn, phương thức này chấp nhận một chuỗi truy vấn và đối tượng ResultSetExtractor:

**public** **class** **JdbcTemplate** {

**public** <T> T query(**final** String sql, **final** ResultSetExtractor<T> rse) throws DataAccessException {

// Execute query...

}

// Other methods...

}

Code language: PHP (php)

ResultSetExtractor chuyển đổi đối tượng ResultSet – đại diện cho kết quả của truy vấn – thành một đối tượng miền thuộc loại T:

@FunctionalInterface

**public** **interface** **ResultSetExtractor**<**T**> {

T extractData(ResultSet rs) throws SQLException, DataAccessException;

}

Code language: PHP (php)

Spring giảm thêm mã soạn sẵn bằng cách tạo các giao diện gọi lại cụ thể hơn.

Ví dụ: interface RowMapper được sử dụng để chuyển đổi một hàng dữ liệu SQL đơn lẻ thành một đối tượng miền kiểu T.

@FunctionalInterface

**public** **interface** **RowMapper**<**T**> {

T mapRow(ResultSet rs, int rowNum) throws SQLException;

}

Code language: PHP (php)

Để điều chỉnh giao diện RowMapper với ResultSetExtractor mong đợi, Spring tạo lớp RowMapperResultSetExtractor:

**public** **class** **JdbcTemplate** {

**public** <T> **List**<T> query(String sql, RowMapper<T> rowMapper) throws DataAccessException {

**return** result(query(sql, **new** RowMapperResultSetExtractor<>(rowMapper)));

}

// Other methods...

}

Code language: PHP (php)

Thay vì cung cấp logic để chuyển đổi toàn bộ đối tượng ResultSet, bao gồm cả việc lặp qua các hàng, tôi có thể cung cấp logic cho cách chuyển đổi một hàng:

**public** **class** **BookRowMapper** **implements** **RowMapper**<**Book**> {

@Override

**public** Book mapRow(ResultSet rs, int rowNum) throws SQLException {

Book book = **new** Book();

book.setId(rs.getLong("id"));

book.setTitle(rs.getString("title"));

book.setAuthor(rs.getString("author"));

**return** book;

}

}

Code language: PHP (php)

Với trình chuyển đổi này, sau đó tôi có thể truy vấn cơ sở dữ liệu bằng JdbcTemplate và ánh xạ từng hàng kết quả:

JdbcTemplate template = // create template...

template.query("SELECT \* FROM books", **new** BookRowMapper());

Code language: JavaScript (javascript)

Ngoài quản lý cơ sở dữ liệu JDBC, Spring cũng sử dụng các mẫu cho:

* Java Message Service (JMS)
* Java Persistence API (JPA)
* Hibernate (now deprecated)
* Transactions

# **SPRING AOP**

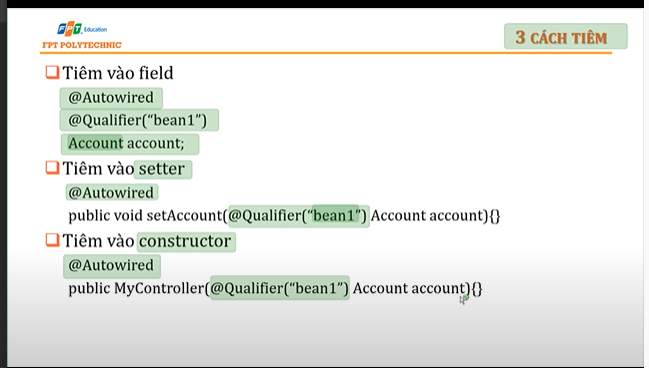
# **Spring core:**

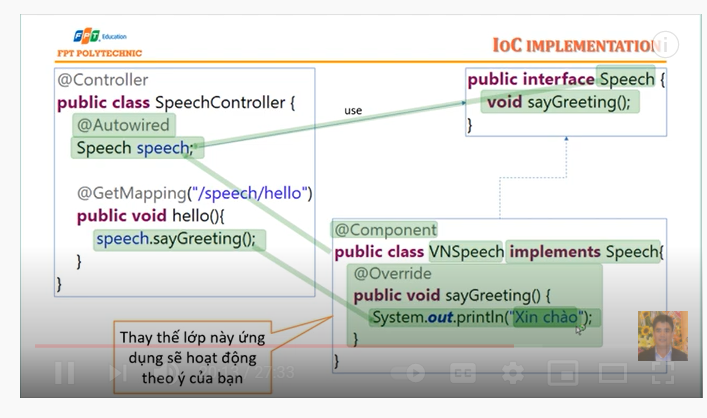
1. IOC, DI, BEAN, CONTAINER

* IOC ~ DI: quá trình đối tượng định nghĩa ra sự thụ thuộc của nó

1. **POJO và JAVABEAN**

**https://shareprogramming.net/pojo-va-javabean-khac-nhau-nhu-the-nao/**

* **POJO ( Plain Old Java Object)**
* Là một class Java Không chịu sự chi phối của 1 framework cụ thể nào
* Không có 1 quy tắc đặt tên cụ thể nào cho thuộc tính và phương thức
* Có thể gặp rắc rối khi sử dụng trong 1 số framework ( POJO Reflection)
* **JAVA BEAN**
* Vẫn là 1 POJO nhưn nó tuân thủ các quy tắc sau:
  + Access level: các thuộc tính đều phải được đặt là private và chỉ được truy xuất thông qua getter and setter
  + Các phương thức getter and setter phải được tuân thủ theo các quy tắc getX(), setX() ( trong đó X là tên của các thuộc tính). Trường hợp phương thức có kiểu trả về là boolean thì có thể đặt là isX**()**
  + Luôn tồn tại 1 contructor mặc định không có tham số đầu vào
  + Implement Serializable Interface
* Hạn chế khi sử dụng JAVA BEAN:
* Mutability: Các Java Bean không thể đạt được tính bất biến (immutable) vì chúng phải định nghĩa các method setter và getter
* BoilerPlate: việc triển khai toàn bộ setter và getter cho các thuộc tính đôi khi la không cần thiết
* No –Arg Constructure: một số thuộc tính yêu cầu phải khởi tạo giá trị
* **SPING BEAN**
* Là các đối tượng (của các lớp )trong java mà được chứa trong container và do container điều khiển (khi khai báo và tiêm sự phụ thuộc)
* Gồm 2 loại: Bean hệ thống, Bean do người dùng tự tạo
* ID: liên kết đến Bean cần thiết
* ICO container: 1 container chứa các bean , phụ trách nạp bean từ bên ngoài vào hệ thống và inject Bean khi cần gọi đến chúng ( quản lý Bean)
* Bean do người dùng định nghĩa: 2 cách
  + + Sử dụng anotation @Autowrite, @Component, @Service, @Repository…
  + + sử dụng @Bean trên phương thức, sẽ trả về 1 instance của lớp đó ( khai báo trong lớp Config với anotation @Configuration
* Trường hợp có nhiều hơn 1 Bean phù hợp với biến hệ thống sẽ báo lỗi ( singleton)🡪 giải quyết: ( set Bean chính với từ khóa Primary, bean khác với @Qualifier(name =”Bean Name”) trong trường hợp sử dụng @Bean
* Tiêm các thể hiện :
* 
* Xây dựng Utilities Bean..

IOC: 

1. **ANOTATION**

### @Transaction:

* there are two distinct ways to configure Transactions, annotations and AOP
* if we're using a Spring Boot project and have a spring-data-\* or spring-tx dependencies on the classpath, then transaction management will be enabled by default.
* At a high level, **Spring creates proxies for all the classes annotated with @Transactional**, either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.
* if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted. **Any self-invocation calls will not start any transaction,**even if the method has the @Transactional annotation.
* Another caveat of using proxies is that **only public methods should be annotated with @Transactional.** Methods of any other visibilities will simply ignore the annotation silently as these are not proxiedTransaction là gì?

Transaction quản lý những thay đổi mà bạn thực hiện trong một hoặc nhiều hệ thống, nó có thể database, message brokers, hoặc bất kỳ loại hệ thống phần mềm nào khác. Mục tiêu chính của giao dịch là cung cấp các đặc điểm [ACID](https://shareprogramming.net/tra-loi-cau-hoi-acid-va-transaction-la-gi-trong-dbms/) để đảm bảo tính nhất quán và hợp lệ của dữ liệu của bạn.

ACID transactions

Vốn dĩ một transaction được đặc trưng bởi 4 yếu tố (thường được gọi là ACID):

* **Atomicity** quy định rằng tất cả các hoạt động của transaction hoặc là thực thi thành công hết hoặc là không có bất cứ hành động nào được thực khi có bất kỳ một hoạt động thực thi không thành công.
* **Consistency** nghĩa là tất cả các ràng buộc toàn vẹn dữ liệu(constraints, key, data types, Trigger, Check) phải được thực thi thành công cho mọi transaction phát sinh xuống database, nhầm đảm bảo tính đúng đắn của dữ liệu.
* **Isolation** đảm bảo các transaction xảy ra xen kẽ sẽ không làm ảnh hưởng đến tính nhất quán của dữ liệu. Các thay đổi dữ liệu bên trong mỗi transaction sẽ được cô lập, các transaction khác sẽ không thể nhìn thấy cho đến khi nó được đồng bộ xuống database.
* **Durability** đảm bảo một transaction thực thi thành công thì tất cả những thay đổi trong transaction phải được đồng bộ xuống database kể cả khi hệ thống xảy ra lỗi hoặc bị mất điện.

Quản lý transaction trong JDBC

* Có 3 thao tác chính bạn có thể thực hiện thông qua java.sql.Connection để kiểm soát transaction trên database.

try (Connection con = dataSource.getConnection()) {

con.setAutoCommit(**false**);

// do something ...

con.commit();

} **catch** (SQLException e) {

con.rollback();

}

* Bạn phải:
  + Bắt đầu transaction bằng cách lấy một Connection và tắt tính năng auto-commit. Điều này cho phép chúng ta kiểm soát transaction. Nếu không, từng câu lệnh SQL sẽ được thực hiện trong các transaction riêng biệt.
  + Commit một transaction bằng cách gọi phương thức commit () trên Connection interface. Điều này khiến database thực hiện tất cả các kiểm tra tính nhất quán cần thiết và lưu các dữ liệu được thay đổi bởi transaction.
  + Rollback (Khôi phục) tất cả các hoạt động trên một transaction bằng cách gọi hàm rollback(). Chúng ta thường thực hiện thao tác này nếu một câu lệnh SQL bị lỗi hoặc nếu bạn phát hiện ra lỗi trong logic nghiệp vụ của mình.
* Quản lý transaction trong Spring
  + quản lý transaction trong JDBC khiến chúng ta phải lặp đi lặp đi các công việc như start, commit, rollback transaction.
  + Spring cung cấp cơ chế hỗ trợ quản lý transaction tự động start, commit, hay rollback transaction tự động.
  + chỉ cần chú thích một class, method hay interface với @Transactional annotation thì chúng sẽ được thực thi bên trong một transaction.
  + Nếu bạn đang sử dụng Spring Boot và có spring-data hay spring-tx dependency thì cơ chế quản lý transaction của Spring sẽ được kích hoạt mặc định.

@Service

public **class** AuthorServiceImpl **implements** AuthorService {

@Autowired

private AuthorRepository authorRepository;

@Override

@Transactional

public **void** updateAuthorName(Long id, String name) {

Author author = authorRepository.findById(id).get();

author.setName(name);

}

}

* @Transactional annotation sẽ nói với Spring rằng updateAuthorName() method cần được thực thi bên trong một transaction. Khi bạn sử dụng AuthorService ở một nơi nào nó, chẳng hạn như ở các controller class, Spring sẽ tạo ra một proxy object bao bọc AuthorService object và cung cấp các đoạn mã cần thiết để bắt đầu một transaction.
* Mặc định, proxy sẽ start một transaction trước khi có một yêu cầu đến method được chú thích với @Transactional annotation. Sau khi method thực thi xong, proxy sẽ commit hoặc rollback transaction nếu có một RuntimeException hoặc Error xảy ra trong quá trình thực thi. Mọi thứ xảy ra ở giữa, chỉ là các đoạn mã code thực thi logic business do chính chúng ta viết.
* @Transactional annottion còn hỗ trợ cho chúng ta tuỳ biến một các hành vi của một transaction thông qua một số thuộc tính quan trọng

https://shareprogramming.net/quan-ly-transaction-voi-spring-va-jpa/

Mỗi một entity phải có một khoá chính dùng để định danh, @Id annotation được dùng để các thuộc tính là khoá chính của table@ENTITY

LINK:<https://shareprogramming.net/dinh-nghia-mot-entity-co-ban-trong-hibernate-jpa/>

CONCEPT:

* Trong JPA – Hibernate, một entity là một [Java class](https://shareprogramming.net/object-va-class-trong-java/) tương ứng với một table trong database, mỗi entity object tương ứng với một dòng trong bảng. JPA yêu cầu một entity class phải cung cấp đầy đủ các metadata để ánh xạ giữa entity class và table như tên cột, kiểu dữ liệu, khoá chính, khoá ngoại, etc.
* JPA và Hibernate đều cho phép một entity class cung cấp metadata qua 2 cách: [*annotation*](https://shareprogramming.net/annotations-trong-java-voi-vi-du-cu-the/) và xml descriptors.

CẤU HÌNH ENTITY

* Một entity tương ứng với một table dưới DATABASE, vì thế nó phải chứa đầy đủ thông tin như tên bản, khóa chính khóa ngoại, các cột trong bảng.
* Mặc định tên của entity sẽ trùng với tên của class, có thể thay đổi thông qua thuộc tính name trong @Entity annotion
  + ví dụ:
* @Entity(name="student")
* public class Student {
* // fields, getters and setters
* }
* **Note**: Bên dưới Hibernate sử dụng các entity object để triển khai các tính năng của ORM nên entity class không được khai báo với [*final*](https://shareprogramming.net/tu-khoa-final-trong-java-voi-vi-du-cu-the/).

### @Id annotation

* Mỗi một entity phải có một khoá chính dùng để định danh, @Id annotation được dùng để các thuộc tính là khoá chính của table
* Chúng ta có thể sinh giá trị cho khóa chính tự động bằng cách sử dụng [@GeneratedValue với các chiến lược](https://shareprogramming.net/generate-primary-key-trong-hibernate-jpa/) khác nhau. Có 4 chiến lược khác nhau chúng ta có thể chọn là, **AUTO, TABLE, SEQUENCE, và IDENTITY.**
* Ví dụ
* @Entity(name = "student")
* public class Student {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* }
* Ở trên mình chọn **IDENTITY,**tuỳ chọn này sẽ tạo ra một khoá chính số nguyên tăng dần mỗi khi một dòng dữ liệu mới được thêm vào. Chúng ta không cần quan tâm đến giá trị của khoá chính mà database sẽ chịu trách nhiệm sinh ra giá trị này.

### @Table annotation

* Mặc định **tên table dưới database(chú ý giúp mình chỗ này)** cũng sẽ giống với Entity class. **Trong trường hợp tên table và tên entity không giống nhau**, chúng ta có thể sử dụng @Table annotation để chỉ rõ table tương ứng với entity name.
* @Entity
* @Table(name="STUDENT")
* public class Student {
* // fields, getters and setters
* }
* @Entity(name = ‘EntityName’) dùng để chỉ tên của entity được Hibernate quản lý . Ví dụ khi chúng ta sử dụng HQL để truy vấn thì chúng ta cần chỉ định **EntityName** và Hibernate sẽ dựa vào đó để ánh xạ thành ***TableName*** tương ứng.
  + Ví dụ: Hibernate: select m from EntityName m => SQL: select m from TableName m
* @Table(name = “TableName”) chỉ đích danh tên của table dưới database

### @Column annotation

* @Column dùng để mô tả các thông tin của một cột trong database bao gồm name, length, nullable, unique.

Ví dụ @Column(name="STUDENT\_NAME", length=50, nullable=**false**, unique=**false**)

private String name;

### @Transient annotation

* @Transient dùng để đánh dấu một thuộc tính trong entity class không phải là một cột tương ứng trong database. Ví dụ chúng ta có thuộc tính **age** được tính bằng công thức năm hiện tại trừ đi năm sinh, do thuộc **age** không cần lưu trong database.
* @Transient
* private Integer age;

### @Enumerated annotation

* @Enumerated được dùng với các cột kiểu [Enum](https://shareprogramming.net/tu-dinh-nghia-enum-trong-java/) trong Java.

### @ComponentScan

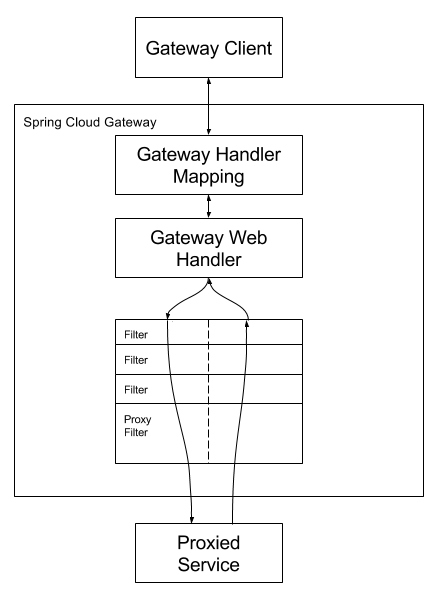
* Đối với các ứng dụng Spring Boot, @SpringBootApplication annotation được sử dụng ở main class là sự kết hợp của cả 3 annotation:
  + @Configuration
  + @EnableAutoConfiguration
  + @ComponentScan
* Như vậy, mặc định Spring Boot sẽ quét tất cả các class ở cùng package ở main class và tất cả các sub-package của nó để tìm và khởi tạo các bean tương ứng đã được định nghĩa.

# **SPRING CLOUD GATEWAY**

1. ***Các thuật ngữ:***

* Route (định tuyến): Khối cơ bản của gateway, được xác định bởi ID, URI đích, 1 tập thuộc tính và 1 tập bộ lọc. 1 route được match nếu tổng hợp các thuộc tính trùng khớp
* Predicate(thuộc tính): là chức năng được thêm vào trong java8, kiểu dữ liệu đầu vào là 1 [Spring Framework ServerWebExchange](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/ServerWebExchange.html), cho phép match bất kỳ thứ gì từ HTTP Request như là header hoặc parameter.
* Filter(Bộ lọc): là mộ thể hiện của  [Spring Framework GatewayFilter](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/GatewayFilter.html) được xây dựng từ các factory xác định. Tại đây ta có thể chỉnh sửa request và response trước và sau khi gửi xuống luồng dữ liệu

1. ***Cơ chế làm việc***



*Hình: cơ chế làm việc của spring cloud gateway*

* Client gửi request tới SCW, nếu Gateway Handler Mapping xác định được request khớp với định tuyến nào nó sẽ được gửi tới Gateway Web Handle(GWH)
* GWH thực hiện chạy request qua nhiều lần lọc để xác định được request, lý do phải lọc qua nhiều lần để bộ lọc có thể chạy logic cả trước và sau trước khi proxy request được gửi đi
* Khi bộ pre-filter logic được thực hiện xong sẽ sinh ra proxy request. Sau đó post filter logic sẽ được thực hiện

1. ***Cấu hình Route Predicate Factories và Gateway Filter Factories***

* Dạng viết tắt:

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: https://example.org

predicates:

- Cookie=mycookie,mycookievalue

- Dạng đầy đủ:

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: https://example.org

predicates:

- name: Cookie

args:

name: mycookie

regexp: mycookievalue

1. ***Route predicate Factories***

* *The After Route Predicate Factory* : nhận 1 tham số là 1 giá trị kiểu datatime. Predicate sẽ match tất cả các request xảy ra sau thời điểm này:

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: https://example.org

predicates:

- After=2017-01-20T17:42:47.789-07:00[America/Denver]

* *The Before Route Predicate Factory:* nhận 1 tham số có giá trị kiểu datetime. Precidatce sẽ match tất cả các request xảy ra trước thời điểm này.

spring:

cloud:

gateway:

routes:

- id: between\_route

uri: https://example.org

predicates:

- Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]

* *The Between Route Predicate Factory:* nhận 2 tham số *datetime1 và datetime2*  có kiểu datetime, trong đó *datetime1< datetime2.* Predicate sẽ match tất cả các request xảy ra trong khoảng thời gian từ datetime1 đến datetime2.

spring:

cloud:

gateway:

routes:

- id: between\_route

uri: https://example.org

predicates:

- Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]

* *The Cookie Route Predicate Factory:* nhận 2 tham số làcookie name và regex:

spring:

cloud:

gateway:

routes:

- id: cookie\_route

uri: https://example.org

predicates:

- Cookie=chocolate, ch.p

- *The Header Route Predicate Factory:* nhận 2 tham số là header-name và regex:

spring:

cloud:

gateway:

routes:

- id: header\_route

uri: https://example.org

predicates:

- Header=X-Request-Id, \d+

- The Host Route Predicate Factory: nhận danh sách các mẫu hostname

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: https://example.org

predicates:

- Host=\*\*.somehost.org,\*\*.anotherhost.org

- The Method Route Predicate Factory: nhận tham số là cá HTTP method:

spring:

cloud:

gateway:

routes:

- id: method\_route

uri: https://example.org

predicates:

- Method=GET,POST

- The Path Route Predicate Factory: nhận 2 tham số là list Spring PathMatcher patterns  và 1 cờ tùy chọn và 1 cờ tùy chọn matchOptionalTrailingSeparator:

spring:

cloud:

gateway:

routes:

- id: path\_route

uri: https://example.org

predicates:

- Path=/red/{segment},/blue/{segment}

- The Query Route Predicate Factory: nhận 2 tham số là required param và optional regex :

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: https://example.org

predicates:

- Query=green, red, gree

- The RemoteAddr Route Predicate Factory: nhận danh sách source IP: spring:

cloud:

gateway:

routes:

- id: remoteaddr\_route

uri: https://example.org

predicates:

- RemoteAddr=192.168.1.1/24

- The Weight Route Predicate Factory: nhận 2 tham số group và weight (dạng int):

spring:

cloud:

gateway:

routes:

- id: weight\_high

uri: https://weighthigh.org

predicates:

- Weight=group1, 8

- id: weight\_low

uri: https://weightlow.org

predicates:

- Weight=group1, 2

1. ***Zuul: Router and Filter***

* Zuul là một thành phần trong kiến trúc của microservice được xây dựng và phát triển bởi neflix đóng vài tròng như một gateway
* Zuul proxy được phát triển thành 1 thành phần của Spring cloud thường được sử dụng trong trường hợp một ứng dụng UI muốn ủy quyển gọi (proxy call) tới một hoặc nhiều backend service, tránh việc cần thiết phải kiểm soát CORS và các vấn đề phụ thuộc liên quan đến xác thực
* zuul.host.maxTotalConnections : default 200
* zuul.host.maxPerRouteConnections: default 20
* zuul.ignored-services: loại bỏ tính năng các service tự động được thêm vào danh sách *service id,* nếu giá trị = “\*” sẽ loại trừ tất cả các service , ngoại trừ các service được khai báo
* zuul:
* routes:
* users: /myusers/\*\* #Các request gửi theo endpoit /myusers sẽ được users service và loại bỏ prefix “/myusers”
* zuul:
* routes:
* users:
* path: /myusers/\*\* #các request có pattern này sẽ được forward đến service và bỏ "/myusers”, /\* là một level, /\*\* là nhiều level)
* serviceId: users\_service #tên serivce
* zuul:
* routes:
* users:
* path: /myusers/\*\* (request pattern)
* url: <https://example.com/users_service> (xác định url request sẽ được forward tới)
* F
* **Sử dụng hystrix command và load-balance multi URL với Ribbon**
* zuul:
* routes:
* echo:
* path: /myusers/\*\*
* serviceId: myusers-service
* stripPrefix: **true** #Enable hystrix
* hystrix:
* command:
* myusers-service: #service id được khai báo ở trên
* execution:
* isolation:
* thread:
* timeoutInMilliseconds: ...
* myusers-service:
* ribbon:
* NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
* listOfServers: https://example1.com,http://example2.com
* ConnectTimeout: 1000
* ReadTimeout: 3000
* MaxTotalHttpConnections: 500
* MaxConnectionsPerHost: 100
* **Cấu hình Ribbon cho list server (tuy nhiên cần off cấu hình của Eureka)**
* zuul:
* routes:
* users:
* path: /myusers/\*\*
* serviceId: users
* ribbon: # off ribbon của eureka
* eureka:
* enabled: **false**
* users: # tên serviceId được khai báo
* ribbon:
* listOfServers: example.com,google.com
* zuul.prefix : thêm prefix cho tất cả các request
* zuul.stripPrefix: nếu gia trị là false thực hiện lạo bỏ dấu “/” trong prefix
* zuul:
* routes:
* users:
* path: /myusers/\*\*
* stripPrefix: **false** #không loại bỏ “/myusers” khi forward req to user serivce
* ***F***
* zuul:
* ignoredPatterns: /\*\*/admin/\*\* #req pattern sẽ không được xử lý
* routes:
* users: /myusers/\*\*
* #ví dụ “/myusers/101” sẽ được forward tới “/101” tuy nhiên “/myusers/admin/101” sẽ không được xử lý
* ***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***Cookies và Sensitive Header:***

* Trong trường hợp không muốn chia sẻ header giữa các service ta có thể loại bỏ chung: Mặc định các thành phần Cookie,Set-Cookie,Authorization được cho vào blacklist, để forward chúng đi được ta cần chỉnh lại tham số sensitiveHeaders
* zuul:
* routes:
* users:
* path: /myusers/\*\*
* sensitiveHeaders: Cookie,Set-Cookie,Authorization
* url: https://downstream
* zuul.sensitiveHeaders: khai báo cho global, nếu khai báo theo từng route thì sẽ được override lại
* ***Ignored Header***
* ignoredHeaders: sẽ có giá trị mặc định là rỗng, để thiết lập các tham số trong header không được gửi đi ta cần set giá trị cho tham số này
* ***Strangulation Patterns and Local Forwards***
* zuul:
* routes:
* first: #req path “/first” được forward đến url
* path: /first/\*\*
* url: http://first.example.com
* second: #req path “/second” được forward và được xử lý nội bộ ví dụ như 1 hàm có chứa @RequestMapping (“/second”)
* path: /second/\*\*
* url: forward:/second
* third:
* path: /third/\*\*
* url: forward:/3rd
* legacy:
* path: /\*\*
* url: http://legacy.example.com
* zuul.host.connect-timeout-millis: set thời gian timeout cho các request
* zuul.host.socket-timeout-millis

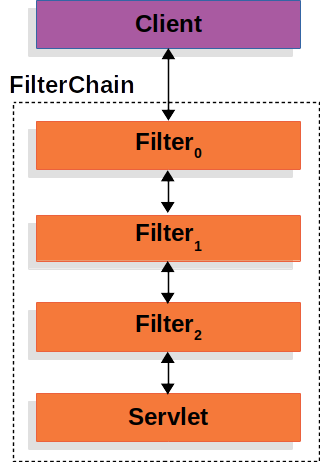
# **SPRING SECURITY**

1. ***Feature***
   1. ***Authenticaton***

* Spring cung cấp 1 inteface PasswordEncode để thực hiện mã hóa password theo 1 chiều
  1. ***Protection Against Exploits***

1. ***Servlet Application***
   1. ***Getting Started***

* Khi thực hiện enable @EnableWebSecurity thì Spring Boot sẽ tự động:
  + Tạo một servlet filter với 1 bean tên *springSecurityFilterChain,* bean này thực hiện tất cả các nhiệm vụ của việc bảo mật trong ứng dụng
  + Tạo một bean UserDetailService với username của “user” và 1 password được show trên mà hình console
  + Đăng ký Filter với 1 bean tên *springSecurityFilterChain* với Servlet container với mỗi request
* Các đặc tính được Spring Security cung cấp:
  + Yêu cầu một sự xác thực với bất cứ sự tương tác nào với ứng dụng
  + Tạo một form login mặc định
  + Tạo một username với “user” và một password mặc định hiển thị trên console
  + Bảo vệc việc lưu trữ password với Bcrypt
  + Cho phép user logout
  + Ngăn chặn tấn công CSRF
  + Bảo vệ việc cố đinh session
  + Tương tác với Security Header
  + Tương tác với các hàm Servlet API
    - [HttpServletRequest#getRemoteUser()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRemoteUser())
    - [HttpServletRequest.html#getUserPrincipal()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getUserPrincipal())
    - [HttpServletRequest.html#isUserInRole(java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#isUserInRole(java.lang.String))
    - [HttpServletRequest.html#login(java.lang.String, java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#login(java.lang.String,%20java.lang.String))
    - [HttpServletRequest.html#logout()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#logout())
  1. ***Kiến trúc***
* Việc hỗ trợ Spring Security’ Sevler chủ yếu dựa trên *Servler Filter*

******

* Khi client gửi req vào ứng dụng container sẽ tạo ra một *FilterChain* có chứa *Filter* và *Servlet (*về bản chất là các HTTPServletRequest và HTTPServletResponse*).*
* Sức mạnh của Filter đến từ *FilterChain* được pass qua nó:

*public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {*

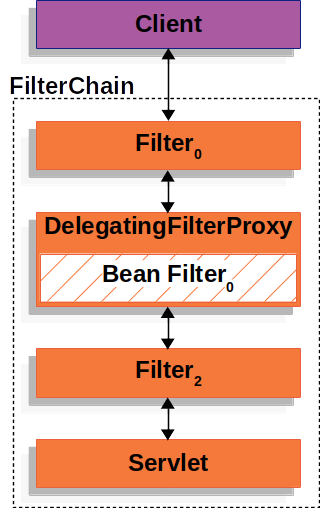
*// do something before the rest of the application*

*chain.doFilter(request, response); // invoke the rest of the application*

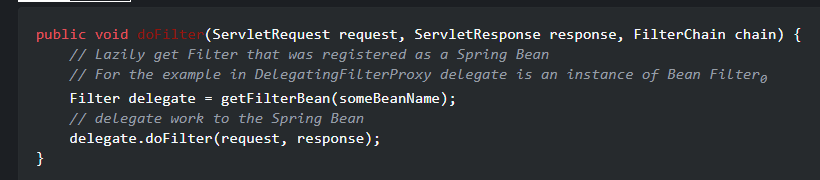
*// do something after the rest of the application*

*}*

* Lưu ý : Filter chỉ tác động xuôi luồng nên việc sắp xếp thứ tự các filter rất quan trọng
* *DelegatingFilterProxy*

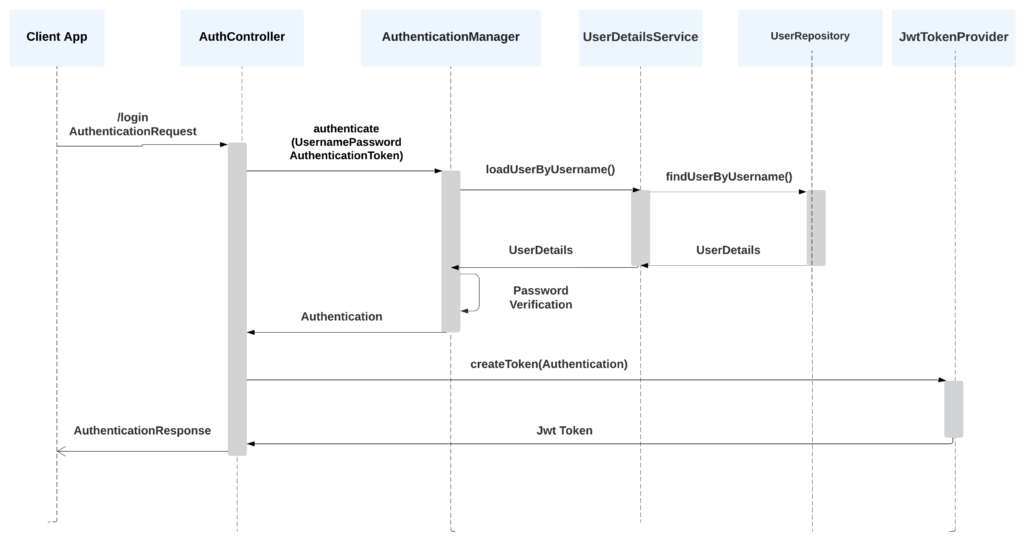


* + *DelegatingFilterProxy* sẽ tìm kiếm BeanFilter0 trong SpringApplicationContext sử dụng bean này

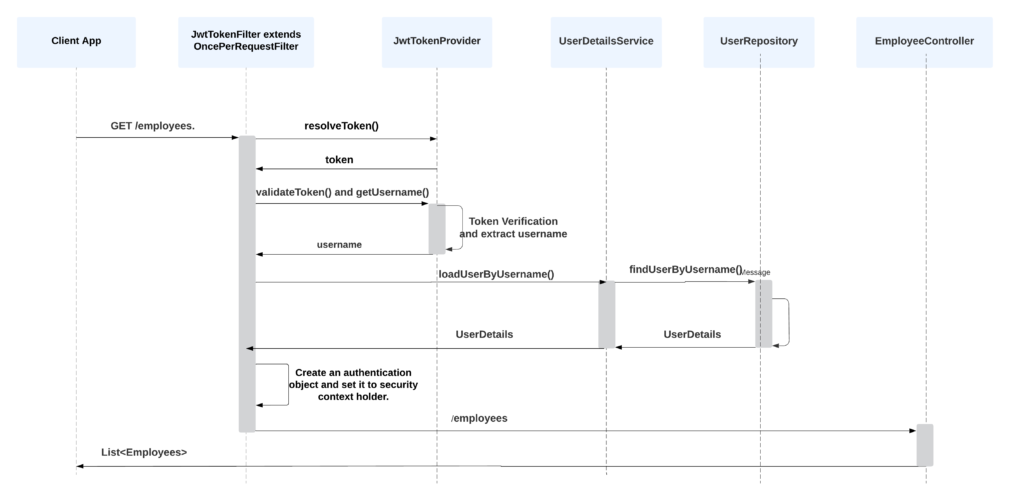


1. ***JWT***

* *Link:* [*https://howtodoinjava.com/spring-security/jwt-auth-vuejs-spring-boot-security/*](https://howtodoinjava.com/spring-security/jwt-auth-vuejs-spring-boot-security/)
  1. *Login Work Flow*



* Client gửi một POST requets tới */login*  với thông tin username và password chứa trong payload
* Hàm *authenticateUser()* trong controller class nhận request và sử dụng  *AuthentiactionManagement* để xác thực user’s cridential. *AuthentiactionManagement* xác nhận thông tin của user bằng cách check username và password có chưa trong dataBase
* Nếu thông tin user được xác thực, sẽ trả về 1 object Authentication, object này có chứa thông tin về user đã được xác thực như: username và các quyền được gán cho user đó
* *SecurityContextHolder* sẽ được cập nhật với object *Authentication* đã được xác thực, do đó ta có thể lấy được thông tin này ở bất cứ đâu trong ứng dụng
* *JwtTokenProvider*  sử dụng *Authentication*  để tạo ra JWT Token cho user đã được xác thực, Token này có thể chứa thông tin username, quyền và thời gian hiệu lực
* Server trả về cho client mã 200 cùng với JSON payload có chứa token, client phải thực hiện lưu token này và gán và header cho cac request tiếp theo
  1. *Authentiaction Workflow*



* *JwtTokenFilter* thực hiện “đánh cắp” request trước khi gửi chúng đến “/employee”
* Lấy JWT Token từ header của request (key = *Authorization*)
* Nếu token được tìm thấy thì thực hiện xác nhận bằng hàm validateToken()
* Nếu token vẫn còn hiệu lực thì thực hiện load thông tin user gắn với token bằng cách sử dụng hàm loadUserByUsername()
* Thực hiện tạo một authentication object (là một kiểu của *UsernamePasswordAuthenticationToken* ) và set vào *SecurityContextHolder*
* Cuối cùng FilterChain được gọi để thực hiện filter tiếp request và response hoặc gửi vào endpoint
  1. *Security Configuration*

@EnableWebSecurity

@Configuration

**public** **class** SecurityConfig {

**private** **final** JwtTokenFilter jwtAuthenticationFilter;

**private** **final** UserDetailsService userDetailsService;

**private** **final** DaoAuthenticationProvider daoAuthenticationProvider;

**public** SecurityConfig(JwtTokenFilter jwtAuthenticationFilter,

UserDetailsService userDetailsService,

DaoAuthenticationProvider daoAuthenticationProvider) {

**this**.jwtAuthenticationFilter = jwtAuthenticationFilter;

**this**.userDetailsService = userDetailsService;

**this**.daoAuthenticationProvider = daoAuthenticationProvider;

}

@Bean

**public** AuthenticationProvider authenticationProvider() {

daoAuthenticationProvider.setUserDetailsService(userDetailsService);

daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());

**return** daoAuthenticationProvider;

}

@Bean

PasswordEncoder passwordEncoder() {

**return** NoOpPasswordEncoder.getInstance();

}

@Bean

AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) **throws** Exception {

**return** authenticationConfiguration.getAuthenticationManager();

}

@Bean

**public** SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) **throws** Exception {

httpSecurity.headers().frameOptions().disable();

httpSecurity.cors().and().csrf().disable();

*//@formatter:off*

httpSecurity

.authorizeHttpRequests()

.requestMatchers("/api/auth/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.exceptionHandling()

.authenticationEntryPoint(

(request, response, authException)

-> response.sendError(

HttpServletResponse.SC\_UNAUTHORIZED,

authException.getLocalizedMessage()

)

)

.and()

.authenticationProvider(authenticationProvider())

.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.**class**);

*//@formatter:on*

**return** httpSecurity.build();

}

}

* *Authentication Management:* thực hiện phối hợp và quản lý luồn xác thực:
  + Cho phép ứng dụng ủy quyền luồng xác thực cho nhiều provider, Mỗi provider sẽ có cơ chế xác thực riêng
  + Khi user thực hiện login, ứng dụng ủy quyền việc xác thực cho *AuthenticationManagement,*  nó sẽ chọn *AuthenticationProvider* thích hợp dựa và kiểu request và forward req đến provider được cấu hình
* *AuthenticationProvider* chỉ xử lý 2 chức năng:
  + *Authenticated()*  để xử lý việc xác thực.
  + *Support()* nếu provider hỗ trợ kiểu xác thực riêng
  + Một interface quan trọng *UserDetailService* có chứa chỉ 1 hàm *loadUserByUsername()* nhận tham số đầu vào là username và trả về thông tin user bằng *UserDeatails*
* *SecurityFilterChain:* là một thành phần quan trọng của SpringSecurity chịu trách nhiệm cho việc filter các cấu hình bảo mật cho các HTTP request

# **SPRING DATA**