I, [core] Khái niệm tight-coupling (liên kết ràng buộc) và cách loosely coupled

1, Giới thiệu

* tight-coupling (liên kết rằng buộc) là một khái niệm trong java ám chỉ việc mối quan hệ giữa các class quá chặt chẽ. Khi yêu cầu thay đổi logic hay một class bị lỗi dẫn tới ảnh hưởng tới toàn bộ các class liên quan đến nó.
* Loosely-coupled là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các class với nhau.

VD:

1, Bạn có một Class thực thi một nhiệm vụ cực kỳ phức tạp, và một trong số đó là việc sắp xếp dữ liệu trước khi xử lý.

public class BubbleSortAlgorithm{

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}

}

public class VeryComplexService {

// tạo mới 1 thằng BubblesortAlgorithm

private BubbleSortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

public VeryComplexService(){

}

public void complexBusiness(int array[]){

bubbleSortAlgorithm.sort(array);

// TODO: logic here

}

}

* Với cách làm ở trên, VeryComplexService đã hoàn thiện được nhiệm vụ, tuy nhiên, khi có yêu cầu thay đổi thuật toán sắp xếp sang QuickSort thì nghe vẻ chúng ta sẽ phải sửa lại hoàn toàn 2 Class ở trên ( sửa logic ở BubbleSortAlgorithm và sửa lại biến ở VeryComplexService )
* Ngoài ra BubbleSortAlgorithm sẽ chỉ tồn tại nếu VeryComplexService tồn tại, vì VeryComplexService tạo đối tượng BubbleSortAlgorithm bên trong nó ( hay nói cách khác là sự sống chết của BubbleSortAlgorithm sẽ do VeryComplexService quyết định ), theo như cách impplement này, nó là liên kết rất chặt với nhau

2,

public interface SortAlgorithm {

/\*\*

\* Sắp xếp mảng đầu vào

\* @param array

\*/

public void sort(int array[]);

}

public class BubbleSortAlgorithm implements SortAlgorithm{

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}

}

public class VeryComplexService {

private SortAlgorithm sortAlgorithm;

public VeryComplexService(){

sortAlgorithm = new BubbleSortAlgorithm();

}

public void complexBusiness(int array[]){

sortAlgorithm.sort(array);

// TODO: more logic here

}

* Với cách làm này, VeryComplexService sẽ chỉ quan hệ với một interface là SortAlgorithm với cách này thì mối quan hệ giảm bớt sự liên kết, nhưng nó không thay đổi được việc thuật toán vẫn đang là BubbleSortAlgorithm

3, cách 3

public interface SortAlgorithm {

\* Sắp xếp mảng đầu vào

public void sort(int array[]);

}

public class BubbleSortAlgorithm implements SortAlgorithm{

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}}

public class QuicksortAlgorithm implements SortAlgorithm {

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật sx nhanh");

}}

public class VeryComplexService {

private SortAlgorithm sortAlgorithm;

public VeryComplexService(SortAlgorithm sortAlgorithm){

this.sortAlgorithm = sortAlgorithm;

}

public void complexBusiness(int array[]){

sortAlgorithm.sort(array);

// TODO: more logic here

}

}

public static void main(String[] args) {

SortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

SortAlgorithm quickSortAlgorithm = new QuicksortAlgorithm();

VeryComplexService business1 = new VeryComplexService(bubbleSortAlgorithm);

VeryComplexService business2 = new VeryComplexService(quickSortAlgorithm);

}

* Cách thứ ba này cũng là cách phổ biến nhất. Mối liên hệ giữa 2 Class đã “lỏng lẻo” hơn trước rấ nhiều. VeryComplexService sẽ không quan tâm đến thuật toán sắp xếp là gì nữa, mà chỉ cần tập trung vào nghiệp vụ. Còn SortAlgorithm sẽ được đưa vào từ bên ngoài tùy theo nhu cầu sử dụng.

II, [core] Giải thích Dependency Injection (DI) và IoC bằng Ngọc Trinh

1, Giới thiệu

* Dependency Injection is a design pattern….
* Có thể hiểu nôm na rằng đó là một phương pháp lập trình, là một thiết kế, template để ta có thể dựa theo để code giúp code clear hơn, dễ hiểu và theo một chuẩn đã được nghiên cứu
* Vậy cuối cùng Dependency Injection là gì?

Vd:

public class Girl{

// tạo 1 class outfit, khai báo nó để sử dụng

private Bikini outfit; // mỗi cô gái sẽ có một bộ bikini khi ra ngoài

public Girl(){

// biến outfit ở đây đang gọi đến class bikini để sử dụng các thuộc tính, phương thức của lớp bikini

outfit = new Bikini(); // Khi bạn tạo ra 1 cô gái, bạn cho cô ta mặc Bikini chẳng hạn

}

}

* Trước hết, qua đoạn code này, bạn sẽ thấy là khi tạo ra một Girl, bạn sẽ tạo ra thêm 1 bộ bikini đi kèm với cô gái đó. Lúc này, Bikini tồn tại mang ý nghĩa là dependency (phụ thuộc) của Girl
* Khi khởi tạo thuộc tính như này, bạn vô tình tạo ra một điểm thắt nút trong chương trình của mình, giả sử, Girl muốn mặc một món đồ khác skirthWithTshirt (Váy + áo) hoặc Nake (Trần chuồng)? hoặc nguy hiểm hơn là Class Bikini bị hỏng ( code lớp Bikini không hoạt động) ?
* Mình sẽ phải sửa lại toàn bộ class or code mới hoàn toàn

2. vấn đề là

* Các Class không nên phụ thuộc vào các kế thừa cấp thấp, mà nên phụ thuộc vào Abtraction (lớp trừu tượng).
* Giải thích:

Vd:

// Một interface cho việc ăn mặc

public interface Outfit {

public void wear();

}

// Một object cấp thấp, implement của Outfits

public class Bikini implements Outfit {

public void wear() {

System.out.println("Đã mặc Bikini");

}

}

// Bây giờ Girl chỉ phụ thuộc vào Outfit. nếu muốn thay đổi đồ của cô gái, chúng ta chỉ cần cho Outfit một thể hiện mới.

// hiểu đơn giản: nếu muốn thay đổi Class bikini ta chỉ cần tạo ra 1 class mới implement từ Interface Outfit là được

public class Girl{

private Outfit outfit;

public Girl(){

outfit = new Bikini();

}

// có thể hiểu khi class girl được tạo ra đồng thời nó đã tự động new 1 lớp gì đó (bikini) như vậy chúng ta không thể thay đổi giá trị truyền vào của lớp girl nữa

}

* Tới đây, chúng ta mới chỉ Abtract hóa thuộc tính của Girl mà thôi, còn thực tế, Girl vẫn đang gắn với một bộ Bikini duy nhất, Vậy muốn thay đồ thì chúng ta phải sửa thêm 1 chút xíu nữa.

Sửa lại code:

public class Girl{

private Outfit outfit;

public Girl(Outfit anything){

this.outfit = anything // Tạo ra một cô gái, với một món đồ tùy biến

// Không bị phụ thuộc quá nhiều vào thời điểm khởi tạo, hay code.

// ở đây outfit là biến truyền vào nên ta có thể thay đổi tùy ý

}

}

public class Main {

public static void main(String[] args) {

Outfit bikini = new Bikini(); // Tạo ra đối tượng Bikini ở ngoài đối tượng

Girl ngocTrinh = new Girl(bikini); // Mặc nó vào cho cô gái khi tạo ra cô ấy.

// new Girl(bikini): đây chính là Injection

}

}

* Với đoạn code ở trên, chúng ta đã gần như tách được bikini ra hoàn toàn khỏi Girl. Điều này làm giảm sự phụ thuộc giữa Girl và Bikini. Mà tăng tính tùy biến, linh hoạt cho code. Bây giờ Girl sẽ hoạt động với Outfit mà thôi. Và Outfit ở đâu ra? Chúng ta tạo ra và đưa nó vào (Injection) cô gái Girl.
* Khái niệm Denpendency Injection từ đây mà ra =vvvvv
* Dependency Injection là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc runtime.
* Bây giờ muốn Girl mặc gì khác, bạn chỉ cần tạo một Class kế thừa Outfit và Inject (tiêm) nó vào Girl là xong.
* Cách để Inject dependency vào một đối tượng có thể kể đến như sau:
* **Constructor Injection**: là cái ở ví dụ, Inject dependency ngay vào Contructor.
* **Setter Injection**: nó sẽ gần giống so với constructor. Vd: girl.setOutfit (new naked())
* **Interface Injection**: mỗi Class muốn inject cái gì, thì phải implement một Interface có chứa hàm inject(xx) (gần như thay thế cho setter). Nếu muốn inject gì đó thì gọi hàm inject(xx) ra. Cách này khó

3, Inversion of Control

* Dependency Injection giúp chúng ta dễ dàng mở rộng code và giảm sự phụ thuộc giữa các dependency với nhau. Tuy nhiên, lúc này, khi code bạn sẽ phải kiêm thêm nhiệm vụ Inject dependency (tiêm sự phụ thuộc). thử tưởng tượng một Class có vài chục dependency thì bạn sẽ phải tự tay inject bao nhiêu lần.
* Gây khó khăn trong quản lý code và denpendency cũng như lúc code

Vd:

public static void main(String[] args) {

Outfit bikini = new Bikini();

Accessories gucci = new GucciAccessories();

HairStyle hair = new KoreanHairStyle();

Girl ngocTrinh = new Girl(bikini, gucci, hair…);

}

* Giá mà có em nào đấy đến inject hộ thì tốt nhỉ :>>
* Bây giờ giả sử, chúng ta định nghĩa trước toàn bộ các dependency có trong Project, mô tả nó và tống nó vào 1 cái kho và giao cho 1 thằng framework quản lý. Bất kỳ các Class nào khi khởi tọa, nó cần dependency gì, thì cái framework đấy sẽ tìm trong kho rồi inject vào đối tượng thay chúng ta. Tiện hơn nhỉ?
* Đó chính là nguyên lý của Inversion of Control (IOC) – đảo chiều sự điều khiển
* Inversion of Control is a programming principle. Flow of control within the application is not controlled by the application it self, but rather by the underlying framework.
* Khi đó, code chúng ta sẽ chỉ cần như này, để lấy ra 1 đối tượng:

@Override

public void run(String... args) throws Exception {

Girl girl = context.getBean(Girl.class);

}

* Đó chính là lý do Spring framework ra đời để thực hiện ý tưởng Inversion of Control (IOC), tuy nhiên, theo thời gian, Spring lớn mạnh và trở thành một hệ sinh thái rộng lớn phục vụ nhiều chức năng trên nền tảng IoC này.

III, [Basic] @Component và @Autowired

1, project

* Cài đặt các gói thư viện của Spring Boot trong Maven bằng cách thêm gọi

spring-boot-starter-parent (là parent của toàn bộ project )

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

* Các thư viện cho lập trình web hoặc server side, chúng ta thêm:

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

* File pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

2, cách chạy ứng dụng

* Nếu trong Java truyền thống, khi chạy cả một project, chúng ta sẽ phải định nghĩa một hàm main() và để nó chạy đầu tiên thì ở đây chúng ta cũng vẫn không thay đổi =>> !!
* Có khác là chúng ta sẽ chỉ cho Spring Boot biết nơi nó khởi chạy lần đầu, để nó cài đặt mọi thứ.
* Cách thực hiện là thêm annotation @SpringBootApplication trên class chính và gọi SpringApplication.run (App.class, args); để chạy project.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args); }}

* Nếu qua bài Dependency Injection (DI) và IoC. Thì có thể hiểu nhiệm vụ của Spring là tạo ra một cái Contianer chứa các Dependency cho chúng ta.
* SpringApplication.run (App.class, args) chính là câu lệnh để tạo ra container. Sau đó nó tìm toàn bộ các denpendency trong project của bạn và đưa vào đó.
* Spring đặt tên cho container là ApplicationContex và dependency là Bean
* Vậy làm sao để Spring biết đâu là dependency? Chúng ta tới với khái niệm @component

3, @Component

* Là một Annotation đánh dấu trên các Class để giúp Spring biết nó là một Bean.

Vd:

* Chúng ta có 1 interface Outfit

public interface Outfit {

public void wear();

}

* Implement nó là Class Bikini

/\*

Đánh dấu class bằng @Component

Class này sẽ được Spring Boot hiểu là một Bean (hoặc dependency)

Và sẽ được Spring Boot quản lý

\*/

@Component

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}

* Chạy chương trình, xem kết quả:

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

ApplicationContext context = SpringApplication.run(App.class, args);

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

// Lấy Bean ra bằng cách

Outfit outfit = context.getBean(Outfit.class);

// In ra để xem thử nó là gì

System.out.println("Instance: " + outfit);

// xài hàm wear()

outfit.wear();

}

}

Output:

[1] Instance: me.loda.spring.helloworld.Bikini@1e1f6d9d

[2] Mặc bikini

* Bạn sẽ thấy Outfit lúc này chính là Bikini. Class được đánh dấu là @Component
* Spring Boot khi chạy sẽ dò tìm toàn bộ các Class cùng cấp được ở trong các package thấp hơn so với Class App mà bạn đã cung cấp cho Spring ( chúng ta có thể cấu hình việc tìm kiếm này, sẽ đề cập sau). Trong quá trình dò tìm này, khi gặp một class được đánh dấu @Component thì nó sẽ tạo ra một instance và đưa vào Applicationcontex để quản lý.

4, @Autowired

* Bây giờ mình tạo ra 1 class Girl và có một thuộc tính là Outfit.
* Mình cũng đánh dấu Girl là một @component. Tức Spring boot cần tạo ra một instance của girl để quản lý.

@Component

public class Girl {

@Autowired

Outfit outfit;

public Girl(Outfit outfit) {

this.outfit = outfit;

}

// GET

// SET

}

* Đánh dấu thuộc tính Outfit của Girl bởi annotation @Autowired. Điều này nói với Spring Boot hãy tự inject (tiêm) một instance của Outfit vào thuộc tính này khi khởi tạo Girl.
* Chạy chương trình:

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

ApplicationContext context = SpringApplication.run(App.class, args);

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

// Lấy Bean ra bằng cách

Outfit outfit = context.getBean(Outfit.class);

// In ra để xem thử nó là gì

System.out.println("Output Instance: " + outfit);

// xài hàm wear()

outfit.wear();

Girl girl = context.getBean(Girl.class);

System.out.println("Girl Instance: " + girl);

System.out.println("Girl Outfit: " + girl.outfit);

girl.outfit.wear();

}

}

Output:

[1] Output Instance: me.loda.spring.helloworld.Bikini@2e16f13a

[2] Mặc bikini

[3] Girl Instance: me.loda.spring.helloworld.Girl@353cb1cb

[4] Girl Outfit: me.loda.spring.helloworld.Bikini@2e16f13a

[5] Mặc bikini

* Spring boot đã tự tạo ra một Girl và trong quá tình tạo ra đó, nó truyền Outfit vào làm thuộc tính

5, Singleton

* Điều đặc biệt là các Bean đươc quản lý bên trong ApplicationContext đề là singleton
* Tức là các class này đều được tạo 1 lần duy nhất trong toàn bộ quá trình chạy

[1] Output Instance: me.loda.spring.helloworld.Bikini@2e16f13a

[4] Girl Outfit: me.loda.spring.helloworld.Bikini@2e16f13a

* Tất cả những Bean được quản lý trong ApplicationContext đều chỉ được tọa ra một lần duy nhất và khi có Class yêu cầu @Autowired thì nó sẽ lấy đối tượng đã được tạo sẵn trong ApplicationContext để inject vào.
* Trong trường hợp muốn mỗi lần sử dụng là một instance hoàn toàn mới. thì hãy đánh dấu @Component đó bằng @Scope(“Prototype”)

@Component

@Scope("prototype")

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}

IV, [Basic] @Autowired - @Primary - @Qualifier

1, giới thiệu

Xml:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

* **Cách inject của spring**
* @Autowired đánh dấu cho spring biết rằng sẽ tự động inject bean tương ứng vào vị trí được đánh dấu

@Component

public class Girl {

// Đánh dấu để Spring inject một đối tượng Outfit vào đây

@Autowired

Outfit outfit;

// public Girl(Outfit outfit) {

// this.outfit = outfit;

// }

// GET

// SET

}

* Sau khi tìm thấy một class đánh dấu @Component. Thì quá trình inject Bean xảy ra theo cách như sau:

1. Nếu Class không có hàm constructor hay setter. Thì sẽ sử dụng java Reflection để đưa đối tượng vào thuộc tính có đánh dấu @Autowired.
2. Nếu có hàm Constructor thì sẽ inject Bean vào bởi tham số của hàm
3. Nếu có hàm setter thì sẽ inject bean vào bởi tham số của hàm

* Như ở vd trên thì đã sử dụng java Reflection để inject Bean vào class Girl. Nếu không sử dụng @Autowired thì sẽ phải có một constructor thay thế, hoặc một setter tương ứng.

@Component

public class Girl {

// Đánh dấu để Spring inject một đối tượng Outfit vào đây

@Autowired

Outfit outfit;

// Spring sẽ inject outfit thông qua Constructor trước

public Girl() { }

// Nếu không tìm thấy Constructor thoả mãn, nó sẽ thông qua setter

public void setOutfit(Outfit outfit) {

this.outfit = outfit;

}

// GET SET}

* Cũng có thể gắn @Autowired lên trên method, thay vì thuộc tính, chức năng cũng vẫn tương tự, nó sẽ tìm Bean phù hợp với method đó và truyền vào.

@Component

public class Girl {

// Đánh dấu để Spring inject một đối tượng Outfit vào đây

Outfit outfit;

// Spring sẽ inject outfit thông qua Constructor trước

public Girl() { }

@Autowired

// Nếu không tìm thấy Constructor thoả mãn, nó sẽ thông qua setter

public void setOutfit(Outfit outfit) {

this.outfit = outfit;

}

// GET

// SET

}

* **VẤN ĐỀ CỦA @AUTOWIRED**
* Trong thực tế, sẽ có trường hợp ta sử dụng @Autowired khi Spring boot có chứa 2 bean cùng loại trong context.
* Lúc này thì spring sẽ bối rối và không biết sử dụng bean nào để inject vào đối tượng.

Vd:

* Class Outfit có 2 kế thừa là Bikini và Naked

public interface Outfit {

public void wear();

}

/\*

Đánh dấu class bằng @Component

Class này sẽ được Spring Boot hiểu là một Bean (hoặc dependency)

Và sẽ được Spring Boot quản lý

\*/

@Component

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}

@Component

public class Naked implements Outfit {

@Override

public void wear() {

System.out.println("Đang không mặc gì");

}

}

* Class Girl yêu cầu inject một Outfit vào cho mình

Output:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

APPLICATION FAILED TO START

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

Parameter 0 of constructor in me.loda.spring.helloprimaryqualifier.Girl required a single bean, but 2 were found:

- bikini: defined in file [/Users/lv00141/Documents/WORKING\_SPACE/GITHUB/spring-boot-learning/spring-boot-helloworld-@Primary - @Qualifier/target/classes/me/loda/spring/helloprimaryqualifier/Bikini.class]

- naked: defined in file [/Users/lv00141/Documents/WORKING\_SPACE/GITHUB/spring-boot-learning/spring-boot …

- đại khái là, trong quá trình cài đặt, nó tìm thấy tới 2 đối tượng thỏa mã Outfit. Giờ nó không biết sử dụng cái nào để inject vào trong Girl

2, **@Primary**

* Cách giải quyết nhứ nhất: @Primary
* @Primary là annotation đnahs dấu trên một bean, giúp nó luôn được ưu tiên lựa chọn trong trường hợp có nhiều bean cùng loại trong context.
* Trong ví dụ trên, nếu chúng ta để Naked là primary. Thì cương trình sẽ chạy bình thường.
* Và hiển nhuw Outfit bên trong Girl sẽ là Naked

@Component

@Primary

public class Naked implements Outfit {

@Override

public void wear() {

System.out.println("Đang không mặc gì");

}

}

* **Chạy thử chương trình:**

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

ApplicationContext context = SpringApplication.run(App.class, args);

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

Girl girl = context.getBean(Girl.class);

System.out.println("Girl Instance: " + girl);

System.out.println("Girl Outfit: " + girl.outfit);

girl.outfit.wear();

}

}

Output:

Girl Instance: me.loda.spring.helloprimaryqualifier.Girl@eb9a089

Girl Outfit: me.loda.spring.helloprimaryqualifier.Naked@1688653c

Đang không mặc gì

* Spring boot đã ưu tiên Naked và inject nó vào Girl.

3, **@Qualifier**- @Qualifier xác định tên của một Bean mà bạn muốn chỉ định inject.

Vd:

@Component("bikini")

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}

@Component("naked")

public class Naked implements Outfit {

@Override

public void wear() {

System.out.println("Đang không mặc gì");

}

}

@Component

public class Girl {

Outfit outfit;

// Đánh dấu để Spring inject một đối tượng Outfit vào đây

public Girl(@Qualifier("naked") Outfit outfit) {

// Spring sẽ inject outfit thông qua Constructor đầu tiên

// Ngoài ra, nó sẽ tìm Bean có @Qualifier("naked") trong context để ịnject

this.outfit = outfit;  
 // GET

// SET

}

V, [Basic] Spring Bean life Cycle + @PostConstruct và @PreDestroy

1, **giới thiệu**

* Trong các bài trước, các bạn đã hiểu các khái niệm cơ bản về Bean và các inject nó trong Spring boot bằng @Component + @Autowired
* Nay chúng ta sẽ tìm hiểu về vòng đời của bean.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

2, **@PostConstruct**

* @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.

@Component

public class Girl {

@PostConstruct

public void postConstruct(){

System.out.println("\t>> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này");

}

}

3, @PreDestroy

* @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.

@Component

public class Girl {

@PreDestroy

public void preDestroy(){

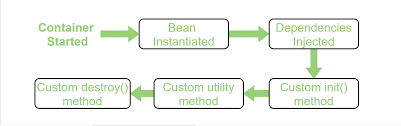
System.out.println("\t>> Đối tượng Girl trước khi bị destroy thì chạy hàm này");

}

}

4, Bean Life Cycle

* Spring Boot từ thời điểm chạy lần đầu tới khi Shutdown thì các Bean nó quản lý sẽ có một vòng đời được biểu diễn như sau:



* Nhìn có vẻ loằng ngoằng, trong series căn bản này, bạn có lẽ sẽ chỉ cần hiểu như sau:

1. Khi IoC Container (ApplicationContext) tìm thấy một Bean cần quản lý, nó sẽ khởi tạo bằng Constructor
2. Inject dependencies vào Bean bằng setter và thực hiện các quá trình cài đặt khác vào Bean như setBeanName, setBeanClassLoader…
3. Hàm đánh dấu @PostConstruct được gọi
4. Tiền xử lý sau khi @PostConstruct được gọi.
5. Bean sẵn sàng được hoạt động
6. Nếu IoC Container không quản lý bean nữa hoặc bị shutdown nó sẽ gọi hàm @PreDestroy trong Bean
7. Xóa Bean.

Vd:

* Chúng ta tạo 1 class Girl bao gồm:

@Component

public class Girl {

@PostConstruct

public void postConstruct(){

System.out.println("\t>> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này");

}

@PreDestroy

public void preDestroy(){

System.out.println("\t>> Đối tượng Girl trước khi bị destroy thì chạy hàm này");

}}

* In ra màn hình quá trình Pring Boot chạy lần đầu tới khi shutdown:

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

System.out.println("> Trước khi IoC Container được khởi tạo");

ApplicationContext context = SpringApplication.run(App.class, args);

System.out.println("> Sau khi IoC Container được khởi tạo");

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

Girl girl = context.getBean(Girl.class);

System.out.println("> Trước khi IoC Container destroy Girl");

((ConfigurableApplicationContext) context).getBeanFactory().destroyBean(girl);

System.out.println("> Sau khi IoC Container destroy Girl");

}

}

Output:

> Trước khi IoC Container được khởi tạo

> Trước khi IoC Container được khởi tạo

>> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này

> Sau khi IoC Container được khởi tạo

> Trước khi IoC Container destroy Girl

>> Đối tượng Girl trước khi bị destroy thì chạy hàm này

> Sau khi IoC Container destroy Girl

* Bạn sẽ thấy dòng “Trước khi IoC Container được khởi tạo” được chạy 2 lần
* Điều này xảy ra bởi vì hàm App.main(args) được chạy 2 lần!
* Lần đầu là do chúng ta chạy
* Lần thứ 2 là do Spring Boot chạy sau khi nó được gọi SpringApplication.run(App.class, agrs). Đây là lúc mà IoC Container (ApplicationContext) được tọa ra và đi tìm Bean.

5, Ý nghĩa

* @PostConstruct và @PreDestroy là 2 Annotation cực kỳ ý nghĩa, nếu bạn nắm được vòng đời của một Bena, bạn có thể tận dụng nó để làm các nhiệm vụ riêng như setting, thêm giá trị mặc định trong thuộc tính sau khi tạo, xóa dữ liệu trước khi xóa, v.v… rất nhiều chức năng khác tùy theo nhu cầu.

VI, [Basic] @component vs @Service vs @Repository

1, Giới Thiệu

* Kiến trúc trong Spring Boot
* Kiến trúc MVC trong Spring Boot được xây dựng dựa trên tư tưởng “độc lập” kết hợp với các nguyên lý thiết kế hướng đối tượng ( một đại diện tiêu biể là Dependency Iversion ). Độc lập ở đây ám chỉ việc các layer phục vụ các mục đích nhất định, khi muốn thwucj hiện một công việc ngoài phạm vi thì sẽ đưa công việc xuống các layer thấp hơn.
* Kiến trúc Controller – Service – Repository chia project thành 3 lớp:

1. **Consumer Layer hay Controller**: là tầng giao tiếp với bên ngoài và handler các request từ bên ngoài hệ thống
2. **Service Layer**: thực hiện các nghiệp vụ và xử lý logic
3. **Repository Layer**: chịu trách nhiệm giao tiếp với các DB, thiết bị lưu trữ, xử lý query và trả về các kieru dữ liệu mà tầng Service yêu cầu



1, **@Controller vs @Service vs @Repository**

* Để phục vụ cho kiến trúc ở trên, Spring Boot tạo ra 3 Annotation là @Controller vs @Service vs @Repository để chúng ta có thể đánh dấu các tầng với nhau.

1. @Service đánh dấu một Class là tầng Service, phục vụ các logic nghiệp vụ
2. @Repository đánh đấu một Class là tầng Repository, phục vụ truy xuất dữ liệu

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.9</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Cấu trúc thư mục:

* Tạo ra một model Girl.

public class Girl {

private String name;

public Girl(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return "Girl(" + this.name + ")";

}

}

* Tạo ra một interface GirlRepository để giao tiếp với DB.

public interface GirlRepository {

/\*\*

\* Tìm kiếm một cô gái trong database theo tên

\* @param name

\* @return

\*/

Girl getGirlByName(String name);

}

* Kế thừa GirlRepository và đánh dấu nó là @Repository

@Repository

public class GirlRepositoryImpl implements GirlRepository {

@Override

public Girl getGirlByName(String name) {

// Ở đây tôi ví dụ là database đã trả về

// một cô gái với tên đúng như tham số

// Còn thực tế phải query trong csđl nhé.

return new Girl(name);

}

}

* Tạo ra một class GirlService để giải quyết các logic nghiệp vụ.
* Lớp GirlService sẽ giao tiếp với DB thông qua GỉlrRepository

import org.apache.commons.lang3.RandomStringUtils;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class GirlService {

@Autowired

private GirlRepository girlRepository;

public Girl getRandomGirl(){

// Random 1 cái tên độ dài 10

String name = randomGirlName(10);

// Gọi xuông tầng repository để query lấy một cô gái tên là "name" trong database

return girlRepository.getGirlByName(name);

}

public String randomGirlName(int length) {

// Random một string có độ dài quy định

// Sử dụng thư viện Apache Common Lang

return RandomStringUtils.randomAlphanumeric(length).toLowerCase();

}

}

* Chạy chương trình:

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

// Lấy ra bean GirlService

GirlService girlService = context.getBean(GirlService.class);

// Lấu ra random một cô gái từ tầng service

Girl girl = girlService.getRandomGirl();

// In ra màn hình

System.out.println(girl);

}

}

Output:

Girl(ulmvchvgkf)

2, Giải Thích

* Về bản chất @Service và @Repository cũng chính là @Component nhưng đặt tên khác nhau giúp chúng ta dễ dàng phân biệt các tầng

Service:

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component // Cũng là một @Component

public @interface Service {

@AliasFor(

annotation = Component.class

)

String value() default "";

}

Repository:

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Repository {

@AliasFor(

annotation = Component.class

)

String value() default "";

}

* Trong các bài đầu tiên chúng ta đã biết @component đánh dấu cho Spring Boot biết class đó là Bean. Và hiển nhiên @Service và @Repository cũng vậy. vì thế ở ví dụ trên chúng ta có thể lấy GirlService từ ApplicationContext.
* Về bản caahst thì có thể sử dụng thay thế cho 3 annotation @Component, @Service, @Repository cho nhau mà không ảnh hưởng gì tới code của bạn cả. nó vẫn sẽ hoạt động.
* Tuy nhiên từ góc độ thiết kế thì chúng ta cần phân rõ 3 annotation này cho các class đảm bảo nhiemj vụ xử lý của nó.

1. @Service gắn cho các Bean đảm nhiệm xử lý logic
2. @Repository gắn cho các Bean đảm nhiệm giao tiếp với DB
3. @Component gắn cho các Bean khác.

VII, [Basic] @Component scan

1, Giới thiệu

* Trong phần trước đã học về 2 annotation thiết kế layout giờ sẽ là annotation thứ 3
* Chúng ta sẽ tìm hiểu về cách Spring Boot tìm kiếm Bean trong project như thế nào.

Pom.xml:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

2, Component scan

* Trong bài 1 có đề cập đến việc Spring boot khi chạy sẽ dò tìm toàn bộ các class cùng cấp hoặc ở trong các package thấp hơn và tạo ra Bean từ các class tìm thấy

Vd:

* Chúng ta có 1 project có cấu trúc như thế này:
* Tạo ra 2 Bean:

1. Girl Nằm cùng package với App
2. OtherGirl nằm ở package con others. Others cùng cấp với App

Girl.java

@Component

public class Girl {

@Override

public String toString() {

return "Girl.java";

}

}

OtherGirl.java

@Component

public class OtherGirl {

@Override

public String toString() {

return "OtherGirl.java";

}

}

App.java

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

try {

Girl girl = context.getBean(Girl.class);

System.out.println("Bean: " + girl.toString());

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

try {

OtherGirl otherGirl = context.getBean(OtherGirl.class);

if (otherGirl != null) {

System.out.println("Bean: " + otherGirl.toString());

}

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

}

}

Output:

Bean: Girl.java

Bean: OtherGirl.java

* Kết quả in ra màn hình là cả 2 bean Girl và OtherGirl đều được tạo ra trong Context.
* Điều này chứng tỏ Spring Boot đã tìm các Bean bên cạnh Class App và những packeage con bên cạnh App

3, @Component scan

* Trong trường hợp muốn tùy hỉnh cấu hình cho Spring boot chỉ cần tím kiếm các bean trong một package nhất định thì có các cách sau đây:

1. Sử dụng @ComponentScan
2. Sử dụng scanBasePackages trong @SpringBootApplication

C1: @ComponentScan

Sửa file app.java thành:

@ComponentScan("me.loda.spring.componentscan.others")

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

try {

Girl girl = context.getBean(Girl.class);

System.out.println("Bean: " + girl.toString());

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

try {

OtherGirl otherGirl = context.getBean(OtherGirl.class);

if (otherGirl != null) {

System.out.println("Bean: " + otherGirl.toString());

}

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

}

}

C2: scanBasePackages

@SpringBootApplication(scanBasePackages = "me.loda.spring.componentscan.others")

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

try {

Girl girl = context.getBean(Girl.class);

System.out.println("Bean: " + girl.toString());

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

try {

OtherGirl otherGirl = context.getBean(OtherGirl.class);

if (otherGirl != null) {

System.out.println("Bean: " + otherGirl.toString());

}

} catch (Exception e) {

System.out.println("Bean Girl không tồn tại");

}

}

}

Output:

Bean Girl không tồn tại

Bean: OtherGirl.java

* Lúc này Spring Boot chỉ tìm kiếm các bean trong package others mà thôi. Nên khi lấy ra Girl thì nó không tồn tại trong context.

4, cấu hình tìm kiếm ở nhiều package

* Có thể cấu hình cho Spring Boot tìm kiếm các Bean ở nhiều package khác nhau bằng cách sau:

@ComponentScan({"me.loda.spring.componentscan.others2","me.loda.spring.componentscan.others"})

Hoặc

@SpringBootApplication(scanBasePackages = {"me.loda.spring.componentscan.others", "me.loda.spring.componentscan.others2"})

VII, [Basic] @configuration và @Bean

1, **trong bài hôm nay ta sẽ tìm hiểu nốt 2 khái niệm cơ bản của Spring Boot**

Pom.xml:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

* Cấu trúc thư mục:

2, **@Configuration và @Bean**

* @Configuration là một annotation đánh dấu trên một Class cho phép Spring Boot biết được đây là nơi định nghĩa ra các Bean.
* @Bean là một annotation được đánh dấu trên các method cho phép spring boot biết được đây là bean và sẽ thực hiện đưa bean này vào Context.
* @Bean sẽ nằm trong các class có đánh dấu @Configuration

Vd:

simpleBeam.java

/\*\*

\* Một class cơ bản, không sử dụng `@Component`

\*/

public class SimpleBean {

private String username;

public SimpleBean(String username) {

setUsername(username);

}

@Override

public String toString() {

return "This is a simple bean, name: " + username;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

}

Appconfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

SimpleBean simpleBeanConfigure(){

// Khởi tạo một instance của SimpleBean và trả ra ngoài

return new SimpleBean("loda");

}

}

App.java

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

// Lấy ra bean SimpleBean trong Context

SimpleBean simpleBean = context.getBean(SimpleBean.class);

// In ra màn hình

System.out.println("Simple Bean Example: " + simpleBean.toString());

}

}

Output:

Simple Bean Example: This is a simple bean, name: loda

* Bạn sẽ thấy là SimpleBean là một object được quản lý trong Context của spring boot, mặc dù trong bài này, chúng ta không hề sử dụng @component

3, In Background

* Đằng sau chương trình, spring boot lần đầu khởi chạy, ngoài việc đi tìm các @Component thì nó còn làm một nhiệm vụ nữa là tìm các class @Configuration.

1. Đi tìm class có đánh dấu @Configuration
2. Tạo ra đối tượng từ class có đánh dấu @Configuration
3. Tìm các method có đánh dấu @Bean trong đối tượng vừa tạo
4. Thực hiện gọi các method có đánh dấu @Bean để lấy ra các Bean và đưa vào Context.

* Ngoài ra, về bản chất, @Configuration cũng là @Component. Nó chỉ khác ở ý nghĩa sử dụng. (giống với việc class đươc đánh dấu @Service chỉ nên phụ vụ logic vậy).

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component // Nó được đánh dấu là Component

public @interface Configuration {

@AliasFor(

annotation = Component.class

)

String value() default "";

}

* Có ý nghĩa gì?
* Nhiều khi tự hỏi rằng @Configuration và @Bean sẽ có ý nghĩa gì khi cúng ta đã có @Component? Sao không đánh dấu SimpleBean là @Component cho nhanh?
* Các bạn thắc mắc hoàn toàn đúng việc sử dụng @Component cũng hoàn toàn ổn.
* Thông thường thì các class được đánh dấu @Component đều có thể tạo tự động và inject tự động được
* Tuy nhiên trong thực tế, nếu một Bean có quá nhiều logic để khởi tạo và cấu hình, thì chúng ta sẽ sử dụng @Configuration và @Bean để tự tay tạo ra Bean. Việc tự tay tạo ra Bean như này có thể hiểu phần nào là chúng ta đang config cho chương trình

Vd:

* Chúng ta sẽ ví dụ với việc cấu hình kết nối tới database. Đây vẫn là một ví dụ hết sức đơn giản.
* Tạo ra một Abstract Class DatabaseConnector chịu trách nhiệm kết nối với database

DatabaseConnector.java

public abstract class DatabaseConnector {

private String url;

/\*\*

\* Hàm này có nhiệm vụ Connect tới một Database bất kỳ

\*/

public abstract void connect();

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

}

* Kế thừa class này có 3 class đại diện:

+ MySqlConnector, PostgreSqlConnnector, MôngDbConnector

MongoDbConnector.java

public class MongoDbConnector extends DatabaseConnector {

@Override

public void connect() {

System.out.println("Đã kết nối tới Mongodb: " + getUrl());

}

}

MySqlConnector.java

public class MySqlConnector extends DatabaseConnector {

@Override

public void connect() {

System.out.println("Đã kết nối tới Mysql: " + getUrl());

}

}

PostgreSqlConnector.java

public class PostgreSqlConnector extends DatabaseConnector{

@Override

public void connect() {

System.out.println("Đã kết nối tới Postgresql: " + getUrl());

}

}

* Tạo ra bean trong AppConfig

AppConfig.java

@Configuration

public class AppConfig {

@Bean("mysqlConnector")

DatabaseConnector mysqlConfigure() {

DatabaseConnector mySqlConnector = new MySqlConnector();

mySqlConnector.setUrl("jdbc:mysql://host1:33060/loda");

// Set username, password, format, v.v...

return mySqlConnector;

}

@Bean("mongodbConnector")

DatabaseConnector mongodbConfigure() {

DatabaseConnector mongoDbConnector = new MongoDbConnector();

mongoDbConnector.setUrl("mongodb://mongodb0.example.com:27017/loda");

// Set username, password, format, v.v...

return mongoDbConnector;

}

@Bean("postgresqlConnector")

DatabaseConnector postgresqlConfigure(){

DatabaseConnector postgreSqlConnector = new PostgreSqlConnector();

postgreSqlConnector.setUrl("postgresql://localhost/loda");

// Set username, password, format, v.v...

return postgreSqlConnector;

}

}

Chạy thử:

App.java

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

DatabaseConnector mysql = (DatabaseConnector) context.getBean("mysqlConnector");

mysql.connect();

DatabaseConnector mongodb = (DatabaseConnector) context.getBean("mongodbConnector");

mongodb.connect();

DatabaseConnector postgresql = (DatabaseConnector) context.getBean("postgresqlConnector");

postgresql.connect();

}

}

Output:

Đã kết nối tới Mysql: jdbc:mysql://host1:33060/loda

Đã kết nối tới Mongodb: mongodb://mongodb0.example.com:27017/loda

Đã kết nối tới Postgresql: postgresql://localhost/loda

* Chúng ta tạo ra DatabaseConnector phục vụ cho nhiều ngữ cảnh

4, @Bean tham số

* Nếu method được đánh dấu bở @Bean có tham số truyền vào, thì Spring Boot sẽ tự inject các Bean đã có trong Context vào làm tham số.

Vd:

AppConfig.java

@Configuration

public class AppConfig {

@Bean

SimpleBean simpleBeanConfigure(){

// Khởi tạo một instance của SimpleBean và trả ra ngoài

return new SimpleBean("loda");

}

@Bean("mysqlConnector")

DatabaseConnector mysqlConfigure(SimpleBean simpleBean) { // SimpleBean được tự động inject vào.

DatabaseConnector mySqlConnector = new MySqlConnector();

mySqlConnector.setUrl("jdbc:mysql://host1:33060/" + simpleBean.getUsername());

// Set username, password, format, v.v...

return mySqlConnector;

}

}

5, thực tế

* Trong thực tế, việc sử dụng @Configuration là hết sức cần thiết , và nó đóng vai trò là nơi cấu hình cho toàn bộ ứng dụng của bạn, một ứng dụng sẽ có nhiều class chứa @Configuratioin và mỗi class sẽ đảm nhận cấu hình một bộ phận gì đó trong ứng dụng.

Vd: đây là một đoạn code cấu hình cho Spring Security

@Configuration

@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/", "/home").permitAll()

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login")

.permitAll()

.and()

.logout()

.permitAll();

}

@Bean

@Override

public UserDetailsService userDetailsService() {

UserDetails user =

User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

return new InMemoryUserDetailsManager(user);

}

}

VIII, [Basic] Spring Boot Application Config và @Value

1, Giới thiệu

* Trong thực tế không phải lúc nòa chúng ta cũng nên để mọi thứ trong code của mình. Có những thông số tốt hơn hết nên được truyền từ bên ngoài vào ứng dụng, để giúp ứng dụng của bạn dễ dàng thay đổi giữa các môi trường khác nhau.
* Để phục vụ điều này, chúng ta sẽ tìm hiểu về khái niệm config ứng dụng spring boot với application.properties

Cài đặt:

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Cấu trúc thư mục:

1**, Application.properties**

* Trong spring boot, các thông tin cấu hình mắc định được lấy từ file resources/applications.properties.

Vd: bạn muốn spring boot chạy trên port 8081 thay vì 8080:

Application.properties

server.port = 8081

* Hoặc bạn muốn log của chương trình chi tiết hơn. Hãy chuyển nó sang dạng debug bằng cách config như sau:

logging.level.root=DEBUG

* Đây là cách chúng ta có thể can thiệp vào các cấu hình của ứng dụng từ bên ngoài. Cho phép thay đổi linh hoạt tùy môi trường.

2, **@Value**

* Trong trường hợp, bạn muốn tự config những giá trị của riêng mình, thì spirng boot hỗ trợ bạn với annotation @Value

Vd: tôi muốn cấu hình cho thông tin database của tôi từ bên ngoài ứng dụng.

Application .properties

loda.mysql.url=jdbc:mysql://host1:33060/loda

* @Value được sử dụng trên các thuộc tính của class, có nhiệm vụ lấy thông tin từ file properties và gán vào biến.

public class AppConfig {

// Lấy giá trị config từ file application.properties

@Value("${loda.mysql.url}")

String mysqlUrl;

}

* Thông tin truyền nào annotation @Value chính là tên của cấu hình đặt trong dấu ${name}

Vd:

* Vẫn là ví dụ trên, chúng ta sẽ làm hoàn chỉnh.
* Thông tin file application.properties bao gồm

Application.properties

server.port = 8081

logging.level.root=INFO

loda.mysql.url=jdbc:mysql://host1:33060/loda

* Tạo ra class DatabaseConnector có nhiệm vụ kết nối tới database.

DatabaseConnector.java

public abstract class DatabaseConnector {

private String url;

/\*\*

\* Hàm này có nhiệm vụ Connect tới một Database bất kỳ

\*/

public abstract void connect();

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

}

Kế thừa nó là MySqlConnector

MySqlConnector.java

public class MySqlConnector extends DatabaseConnector {

@Override

public void connect() {

System.out.println("Đã kết nối tới Mysql: " + getUrl());

}

}

* Chương trình sẽ được cấu hình trong AppConfig

AppConfig.java

@Configuration

public class AppConfig {

// Lấy giá trị config từ file application.properties

@Value("${loda.mysql.url}")

String mysqlUrl;

@Bean

DatabaseConnector mysqlConfigure() {

DatabaseConnector mySqlConnector = new MySqlConnector();

// Set Url

System.out.println("Config Mysql Url: " + mysqlUrl);

mySqlConnector.setUrl(mysqlUrl);

// Set username, password, format, v.v...

return mySqlConnector;

}

}

* Chạy thử chương trình:

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

DatabaseConnector databaseConnector = context.getBean(DatabaseConnector.class);

databaseConnector.connect();

}

}

Ouput:

2019-05-18 17:16:45.489 INFO 14004 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8081 (http) with context path ''

2019-05-18 17:16:45.493 INFO 14004 --- [ restartedMain] m.loda.spring.applicationproperties.App : Started App in 4.402 seconds (JVM running for 5.932)

Đã kết nối tới Mysql: jdbc:mysql://host1:33060/loda

* Bạn sẽ thấy là chương trình đã chạy trên port 8081. Và cấu hình về đường dẫn mysql của tôi tự tạo ra cũng được spring boot đọc lên và đưa vào giá trị này vào biến.

IX, [Basic] tạo web helloword với @Controller + thymeleaf

I, **Giới thiệu**

* Các bài trước chúng ta đã tìm hiểu về các khái niệm cơ bản về spring boot mà ai học cũng đều phải biết.
* Từ bài này, chúng ta sẽ học về các mảng chính mà người ta áp dụng spring boot vào, đó là lập trình **web** hoặc **restful web service**.
* Chúng ta sẽ bắt đầu với controller đầu tiên.

Cài đặt:

* Trong bài này, tôi có sử dụng thêm thư viện spring-boot-starter-thymeleaf. Đây là một **Template engine** hỗ trợ chúng ta tạo ra các file html để trả về thông tin cho ngời dùng. Về cơ bản là như vậy, còn chi tiết tôi sẽ giới thiệu riêng ở bài sau.
* Tạm thời cứ để nó là một thư viện trong chương trình của bạn để chúng ta có thể chạy suôn sẻ.

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- thymeleaf -->

<dependency>

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

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

* Cấu trúc thư mục

2, **@Controller**

* Trong bài số 4, khi nói về @Service và @Repository tôi đã đề cập tới kiến trúc trong **SpringBoot**
* Để xây dựng một trang web với **SpringBoot,** bạn sẽ cần tuân theo quy trình dưới đây:
* @Controller là nơi tiếp nhận các thông tin request từ phía người dùng, nó có nhiệm vụ đson nhận các yêu cầu (kèm theo thông tin request) và chuyển các yêu cầu này xuống cho tầng @Service xử lý logic.

3, **HTML**

* Để tạo ra một trang web, bạn sẽ cần tạo ra các trang html để trả về giao diện cho người dùng.
* Mặc định trong Spring Boot, các file html này sẽ được lưu trữ trong mục resources/templates như sau:

+ **Spring Boot + thymeleaf**  sẽ tìm kiếm các file này theo tên.

Vd: “index” sẽ tương ứng với “index.html”

4, **Hello World**

* Chúng ta sẽ tạo ra một Server web đơn giản để dễ dàng hiểu @Controller làm việc như thế nào.

WebController.java

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

// Đánh dấu đây là một Controller

// Nơi tiếp nhận các reqquest từ phía người dùng

@Controller

public class WebController {

// Đón nhận request GET

@GetMapping("/") // Nếu người dùng request tới địa chỉ "/"

public String index() {

return "index"; // Trả về file index.html

}

}

Index.html

<!DOCTYPE html>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h1>Đây là một trang web</h1>

</body>

</html>

Chạy chương trình:

App.java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

* Lúc này Server đã được chạy trên port 8080.
* Bạn hãy vào trình duyệt theo đường dẫn <http://localhost:8080/>

Output:

* **Giải thích 1:**
* Bản thân @Controller cũng là một @Component nên nó sẽ được Spring Boot quản lý.

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Controller {

@AliasFor(

annotation = Component.class

)

String value() default "";

}

* Spring boot sẽ lắng nghe các request từ phía người dùng. Và tùy theo đường dẫn path là gì, nó sẽ mapping tới hàm xử lý tương ứng trong @Controller.
* Như ví dụ trên, tôi sử dụng GET vào địa chỉ localhost:8080/ ( đường dẫn là / ). **Spring Boot** sẽ gọi tới hàm có gắn @GetMapping(“/”) và yêu cầu hàm này xử lý request này.
* Trong ví dụ trên, tôi chỉ trả về một file index.html là xong, không cần nghiệp vụ gì khác.

Vd2:

* Chúng ta mở rộng thêm một chút, để tạo trang **Hello your name!**
* Cấu trúc thư mục:

Mở rộng WebController như sau:

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

// Đánh dấu đây là một Controller

// Nơi tiếp nhận các reqquest từ phía người dùng

@Controller

public class WebController {

// Đón nhận request GET

@GetMapping("/") // Nếu người dùng request tới địa chỉ "/"

public String index() {

return "index"; // Trả về file index.html

}

@GetMapping("/about") // Nếu người dùng request tới địa chỉ "/about"

public String about() {

return "about"; // Trả về file about.html

}

@GetMapping("/hello")

public String hello(

// Request param "name" sẽ được gán giá trị vào biến String

@RequestParam(name = "name", required = false, defaultValue = "") String name,

// Model là một object của Spring Boot, được gắn vào trong mọi request.

Model model

) {

// Gắn vào model giá trị name nhận được

model.addAttribute("name", name);

return "hello"; // trả về file hello.html cùng với thông tin trong object Model

}

}

Index.html

<!DOCTYPE html>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h1>Đây là một trang web</h1>

<a href="/about">About</a>

<form method="get" action="/hello">

<input type="input" name="name">

<button type="submit">Submit</button>

</form>

</body>

</html>

About.html

<!DOCTYPE html>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h1>Loda</h1>

<a href="https://loda.me">Website</a>

<a href="https://github.com/loda-kun/spring-boot-learning">Github</a>

</body>

</html>

Hello.html

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<title>Hello World</title>

</head>

<body>

<h1 th:text="'Hello, ' + ${name}"></h1>

<a href="/">Trang chủ</a>

</body>

</html>

Chạy chương trình:

App.java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

<http://localhost:8080/>

* Nhập tên của bạn vào ô submit. Nó sẽ tạo một request GET tới địa chỉ /hello kèm theo param ?name=your\_name.

Vd: tôi nhập “Loda”

* Nó sẽ có url là:  <http://localhost:8080/hello?name=Loda>

**Giải thích 2:**

* Tới đây bạn hãy tham chiếu đường dẫn request với hàm xử lý nó:

//http://localhost:8080/hello?name=Loda

@GetMapping("/hello")

public String hello(

// Request param "name" sẽ được gán giá trị vào biến String

@RequestParam(name = "name", required = false, defaultValue = "") String name,

// Model là một object của Spring Boot, được gắn vào trong mọi request.

Model model

) {

// Gắn vào model giá trị name nhận được

model.addAttribute("name", name);

return "hello"; // trả về file hello.html cùng với thông tin trong object Model

}

* khi request lên, chúng ta nhận đợc giá trị của name và tiếp tục gán nó vào Model.
* Model ở đây là một object được **Spring Boot** đính kèm trong mỗi respone.
* Model chứa các thông tin mà bạn muốn trả về và **Template Engine** sẽ trích xuất thông tin này ra thnahf html và đưa cho người dùng.
* Trong file hello.html tôi lấy giá trị của name trong Model ra bằng cách sử dụng cú pháp của Thymeleaf.

<h1 th:text="'Hello, ' + ${name}"></h1>

X, [SB9] giải thích cách Thymeleaf vận hành + Expression + Demo Full

1, **Giới thiệu**

* Trong bài trước, mình đã đề cập với các bạn kiến trúc web của Spring Boot và giới thiệu chức năng của @Controller
* Trong bài này, chúng ta sẽ tìm hiểu chi tiết hơn về **Thymeleaf** và sử dụng nó để làm một trang web đơn giản.

2, **Thymeleaf**

* **Thymeleaf** là một Java Template Engine. Có nhiệm vụ xử lý và genereate ra các file HTML, XML, v.v..
* Các file HTML do Thymeleaf tạo ra là nhờ kết hợp **dữ liệu** và **template + quy tắc** để sinh ra một file HTMl chứa đầy đủ thông tin.
* Việc của bạn là cung cấp dữ liệu và quy định template như nào, còn việc dùng các thông tin đó để render ra HTML sẽ do Thymeleaf để giải quyết.
* Cú pháp.
* Cú pháp của **Thymeleaf** sẽ là một **attributes** ( thuộc tính ) của thẻ HTML và bắt đầu bằng chữ th:
* Với cách tiếp cận này, bạn sẽ chỉ cần sử dụng các thẻ THML cơ bản đã biết mà không cần bổ sung thêm syntax hay thẻ mới như JSP truyền thống

Vd:

* Để truyền dữ liệu từ biến name trong java vào một thẻ H1 của HTML.

<h1 th:text="${name}"></h1>

* Chúng ta viết thẻ H1 như thường, nhưng không chứa bất cứ text nào trong thẻ mà sử dụng cú pháp th:text=”${name}” để Thymeleaf lấy thông tin từ biến name và đưa vào thẻ H1

Ouput:

// Giả sử String name = "loda"

<h1>Loda</h1>

* Thuộc tính th:text biến mất và giá trị biến name được đưa vào trong thẻ H1. Đó là cách mà **thymeleaf** hoạt động

3, **Model & View trong spring bootn**

* Trong bài trước, đã demo cách sử dụng đối tượng Model. Bây giờ tôi sẽ nói kỹ hơn một chút.
* Model là đối tượng lưu giữ thông tin và được sử dụng bởi **Template Engine** để genreate ra webpage. Có thể hiểu nó là Context của **Thymeleaf**.
* Model lưu giữ thông tin dưới dạng key-vlue
* Trong template thmeleaf, để lấy các thông tin trong Model. Bạn sẽ sử dụng Thymleaf Standard Expression.

1. ${…}: giá trị của một biến
2. {…}: giá trị của một biến được chỉ định ( ở dưới )

* Ngoài ra để lấy thông tind đặc biệt hơn:

1. #{…}: lấy message
2. @{…}: lấy đường dẫn URL dự theo contexxt của server

* Nói tới đây có lẽ hơi khó hiểu, chúng ta sẽ dùng ví dụ để hiểu rõ từng loại Expresstion
* \*{…} **– Variables Expresssions on selections**
* Dấu \* còn gọi là asterisk syntax. Chức năng của nó giống với ${..} là lấy giá trị của một biến
* Điểm khác biệt là nó sẽ lấy ra giá trị của một biến cho trước bở th:object

<divth:object="${session.user}"><!-- th:object tồn tại trong phạm vi của thẻ div này -->

<!-- Lấy ra tên của đối tượng session.user -->

<p>Name: <spanth:text="\*{firstName}"></span>.</p><!-- Lấy ra lastName của đối tượng session.user -->

<p>Surname: <spanth:text="\*{lastName}"></span>.</p></div>

* Còn ${…} sẽ lấy ra giá trị cục bộ trong Context hay Model.

Đoạn code này tường đương với:

<div><p>Name: <spanth:text=”${session.user.firstName}”></span>.</p><p>Surname: <spanth:text=”${session.user.lastName}”></span>.</p></div>

* #{…} – **Message Expression**

Vd: trong file config .properties của tôi có một message chào người dùng bằng nhiều ngôn ngữ

home.welcome=¡Bienvenido a nuestra tienda de comestibles!

* Thì cách lấy nó nhanh nhất là:

<p th:utext="#{home.welcome}">Xin chào các bạn!</p>

* Đoạn text tiếng việt bên trong thẻ p sẽ bị thay thế bởi thymeleaf khi render #{home.welcome}
* @{…} – **URL Expresssion**
* @{…} xử lý và trả ra giá trị URL theo context của máy chủ cho chúng ta.

Vd:

<!-- tương đương với 'http://localhost:8080/order/details?orderId=3' -->

<ahref="details.html"th:href="@{http://localhost:8080/order/details(orderId=${o.id})}">view</a><!-- tương đương '/order/details?orderId=3' -->

<ahref="details.html"th:href="@{/order/details(orderId=${o.id})}">view</a><!-- tương dương '/gtvg/order/3/details' -->

<ahref="details.html"th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>

* Nếu bắt đầu bằng dấu / thì nó sẽ là Relative URL và sẽ tương ứng theo context của máy chủ của bạn

3, **Demo minh họa**

* Chúng ta sẽ tạo ra một trang web đơn giản để hiểu các loại Expression trong Thymeleaf.

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- thymeleaf -->

<dependency>

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

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Cấu trúc thư mục:

* **Cấu hình thymeleaf**
* Chúng ta sẽ cấu hình ứng dụng bằng file application.properties.

Applocation.properties.

#Chạy ứng dụng trên port 8085

server.port=8085

# Bỏ tính năng cache của thymeleaf để lập trình cho nhanh

spring.thymeleaf.cache=false

# Các message tĩnh sẽ được lưu tại thư mục i18n

spring.messages.basename=i18n/messages

# Bỏ properties này đi khi deploy

# Nó có tác dụng cố định ngôn ngữ hiện tại chỉ là Tiếng Việt

spring.mvc.locale-resolver=fixed

# Mặc định ngôn ngữ là tiếng việt

spring.mvc.locale=vi\_VN

# Đổi thành tiếng anh bằng cách bỏ comment ở dứoi

# spring.mvc.locale=en\_US

* **Chức năng messages**
* các trang web hỗ trợ đa ngôn ngữ (i18n) thì các message sẽ được lưu dưới dạng key-value. Và tùy theo từng vùng địa lý mà chọn sử dụng value cho hợp lý.
* **Thymeleaf** sẽ tự làm điều này cho chúng ta, vì ở trên chúng ta đã cấu hình cho nó vị trí lưu trữ các messages này rồi.

Vd:

* Tôi có một câu chào đơn giản cho 2 vùng là US và VN.

I18n/messages\_vi.properties

loda.hello=Xin chào tất cả các bạn tới với Loda Website

i18/messages\_en.properties

loda.hello=Welcome to Loda Website

* **Static & Templates**
* Làm web thì không thể thiếu Css và Javascript. Các file này sẽ được lưu tại resources/static.
* File .html là dạng template sử dụng để render re webpage và trả về cho người dùng. Nó sẽ được lưu tại thư mục resources/templates
* **Thymleaf** sẽ tự biết tìm đường tới những tài nguyên này
* Index.html
* Index.html sẽ là file mặc định mà **Thymeleaf** tìm đầu tiên và trả về mỗi khi người dùng vào địa chỉ / hay <https://localhost:8085/> mà chúng ta không cần config gì cả.
* Trong index.html tôi sẽ:

1. Gọi ra bootstrap.css và bootstrap.js trong thư mục resources/static bằng expression @{…}
2. Hiển thị dòng chữ chào loda.hello trong thư mục resources/i18n bằng expression #{…}

Index.html

<!DOCTYPE html>

<htmlxmlns:th="http://www.thymeleaf.org"><head><metacharset="UTF-8"/>

<title>Loda</title><metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><!--css-->

<linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><!--js-->

<scriptth:src="@{/js/bootstrap.js}"></script>

</head><body><h1th:utext="#{loda.hello}"></h1>

<ath:href="@{/profile}"class="btn btn-primary">Loda Profile</a>

</body>

</html>

* **Chạy thử lần 1**.
* sau khi config và tạo ra file index.html là chúng ta đã có một trang web cơ bản rồi.

App.java

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

* truy cập vào địa chỉ <http://localhost:8085/>
* ứng dụng hiện ra dòng chữ chào mừng lấy từ messages và button được trang trí nhờ bootstrap trong static.
* **Thêm @Controller cho path/profile**
* Bây giờ chúng ta sẽ bổ sung thêm @Controller để handle các request tới địa chỉ /profile.

Info.java

import lombok.AllArgsConstructor;

import lombok.Data;

/\*\*

\* TẠo ra class này chỉ để lưu giữ thông tin

\*/

@Data // không khuyến khích cho lắm

@AllArgsConstructor

public class Info {

String key;

String value;

}

webController.java

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

@Controller

public class WebController {

@GetMapping("/profile")

public String profile(Model model){

// Tạo ra thông tin

List<Info> profile = new ArrayList<>();

profile.add(new Info("fullname", "Nguyễn Hoàng Nam"));

profile.add(new Info("nickname", "lốddaf"));

profile.add(new Info("gmail", "loda.namnh@gmail.com"));

profile.add(new Info("facebook", "https://www.facebook.com/nam.tehee"));

profile.add(new Info("website", "https://loda.me"));

// Đưa thông tin vào Model

model.addAttribute("lodaProfile", profile);

// TRả về template profile.html

return "profile";

}

}

* Sau đó chúng ta tạo ra thêm template profile.html để xử lý các thông tin trong Model.

Resources/templates/profile.html

<!DOCTYPE html>

<htmlxmlns:th="http://www.thymeleaf.org">

<head><metacharset="UTF-8"/>

<title>Loda</title>

<metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><!--css-->

<linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><!--js-->

<scriptth:src="@{/js/bootstrap.js}"></script>

</head>

<body><h1th:utext="#{loda.hello}"></h1>

<h2>Loda Profile</h2>

<ul><!--Duyệt qua toàn bộ phần tử trong biến "lodaProfile"-->

<lith:each="info : ${lodaProfile}"><!--Với mỗi phần tử, lấy ra key và value-->

<spanth:text="\*{info.key}">

</span> : <spanth:text="\*{info.value}"></span>

</li>

</ul>

</body>

</html>

* ở đây chúng ta sử dụng:

1. ${…} để lấy giá trị của biến lodaProfile trong Model.
2. {…} để lấy giá trị của biến info đã chỉ định.

* **Chạy thử lần 2.**
* Chúng ta chạy lại chương trình lần nữa và truy cập vào địa chỉ <http://localhost:8085/>.
* Click vào Button Loda Profile. Trình duyệt sẽ đi tới địa chỉ <http://localhost:8085/profile>.

XI, [Basic] @RequestMapping + @PostMapping + @ModelAttribute + @RequestParam + Web To-Do với Thymeleaf

1, **Giới Thiệu**

* bài hôm nay chúng ta sẽ tìm hiểu cách handle request POST của Spring Boot

2,  **@PostMapping**

* @PostMapping có nhiệm vụ đánh dấu hàm xử lý POST request trong Controller.
* Khái niệm về GET và POST hi vọng các bạn đã biết rồi và tôi sẽ không đề cập trong bài này nữa.

Vd:

* Tôi có 2 hàm xử lý, một cho GET method và một cho POST.
* Cả 2 đều chung đường dẫ nhưng bạn nên biết rằng cùng một path nhưng khác method thì sẽ xử lý khác nhau.

@Controller

public class WebController {

@GetMapping("/addTodo")

public String addTodo(Model model) {

return "addTodo";

}

@PostMapping("/addTodo")

public String addTodo(Model model) {

return "success";

}

}

* Từ đây bạn có thể suy ra cách handle cho các method khác như PUT, DELETE, v.v…

1. @PutMapping
2. @DeleteMapping

3, **@RequestMapping**

* Trong trường hợp bạn muốn tất cả các method đều dùng chung một cách xử lý thì có thể sử dụng Annotaiton @RequestMapping.
* @RequestMapping là một annotation có ý nghĩa và mục đích sử dụng rộng hơn các loại @GetMapping, @PostMapping, v.vv…

Vd:

@Controller

@RequestMapping("api/v1")

public class WebController {

// Đường dẫn lúc này là: /api/v1/addTodo và method GET

@RequestMapping(value = "/addTodo", method = RequestMethod.GET)

public String addTodo(Model model) {

return "addTodo";

}

// Đường dẫn lúc này là: /api/v1/addTodo và method POST

@RequestMapping(value = "/addTodo", method = RequestMethod.POST)

public String addTodo(@ModelAttribute Todo todo) {

return "success";

}

}

4, **Website To-do**

* Vẫn còn một kiến thức tôi cần giới thiệu với các bạn.
* Tuy nhiên tôi4sẽ sử dụng ví dụ để giúp bạn hiểu nó nhanh hơn.
* Chúng ta sẽ làm một WebApp quản lý công việc và từ đó tìm hiểu cách gửi dữ liệu lên Server thông qua 2 method GET và POST.

**Cài đặt:**

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- thymeleaf -->

<dependency>

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

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build></project>

Cấu trúc thư mục:

1. Tạo model

* Chúng ta sẽ tạo ra một đối tượng gọi là Todo để thể hiên thông tin.

Todo.java

import lombok.Data;

@Data

public class Todo {

public String title;

public String detail;

}

1. **GET/ listTodo – lấy danh sách các việc cần làm**

* Chúng ta sẽ dùng đường dẫn /listTodo để lấy ra danh sách việc cần làm.
* Thiết kế method cho đường dẫn /listTodo là GET.

webController.java

@Controller

public class WebController {

// Sử dụng tạm List để chứa danh sách công việc

// Thay cho Database.

// Chỉ dùng cách này khi DEMO ^^

List<Todo> todoList = new CopyOnWriteArrayList<>();

/\*

@RequestParam dùng để đánh dấu một biến là request param trong request gửi lên server.

Nó sẽ gán dữ liệu của param-name tương ứng vào biến

\*/

@GetMapping("/listTodo")

public String index(Model model, @RequestParam(value = "limit", required = false) Integer limit) {

// Trả về đối tượng todoList.

// Nếu người dùng gửi lên param limit thì trả về sublist của todoList

model.addAttribute("todoList", limit != null ? todoList.subList(0, limit) : todoList);

// Trả về template "listTodo.html"

return "listTodo";

}}

* Khi tôi request lên server như này <http://localhost:8080/listTodo?limit=2>
* Thì cái đoạn ?limit=2 là RequestParam
* Spring boot sẽ tự xử lý và gán số 2 vào biến Integer limit hộ chúng ta.
* Nếu chúng ta không gửi gì lên, thì limit sẽ là null.

listTodo.html

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head><metacharset="UTF-8"/>

<title>Loda To-do</title><metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><!--css-->

<linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><linkth:href="@{/css/main.css}"rel="stylesheet"/><!--js-->

<scriptth:src="@{/js/bootstrap.js}"></script>

</head>

<body>

<h1>Danh sách việc cần làm:</h1>

<ul><!--Duyệt qua toàn bộ phần tử trong biến "todoList"-->

<lith:each="todo : ${todoList}"><!--Với mỗi phần tử, lấy ra title và detail-->

<spanth:text="\*{todo.getTitle()}">

</span> : <spanth:text="\*{todo.getDetail()}">

</span>

</li>

</ul><ath:href="@{/addTodo}"class="btn btn-success"> + Thêm công việc</a>

</body>

</html>

1. **GET /addTodo – trang thêm công việc**

* GET/addTodo sẽ trả về webpage cho người dùng nhập thông tin công việc và thêm vào danh sách việc cần làm.

webController.java

@GetMapping("/addTodo")

public String addTodo(Model model) {

// Thêm mới một đối tượng Todo vào model

model.addAttribute("todo", new Todo());

// Trả về template addTodo.html

return "addTodo";

}

addTodo.html

<!DOCTYPE html>

<htmlxmlns:th="http://www.thymeleaf.org">

<head

><metacharset="UTF-8"/>

<title>Loda To-do</title>

<metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><!--css-->

<linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><linkth:href="@{/css/main.css}"rel="stylesheet"/><!--js-->

<scriptth:src="@{/js/bootstrap.js}"></script>

</head>

<body>

<h1>To-do</h1>

<formth:action="@{/addTodo}"th:object="${todo}"method="post">

<p>title: <inputtype="text"th:field="\*{title}"/></p>

<p>detail: <inputtype="text"th:field="\*{detail}"/></p>

<p><button type="submit"class="btn btn-success">Add</button></p>

</form>

</body>

</html>

* <form></form> là thẻ tag mà bạn cần biết khi làm web. Nó tạo Request Form gửi tới server với thông tin và các thẻ <input> trong nó.
* ở đây, tôi gán vào Model một đối tượng Todo.
* Trong form, chúng ta lấy ra đối tượng Todo và chỉ định bởi th:object=”${todo}”
* Gán thông tin người dùng vào Todo bằng cú pháp th:field=\*{ten\_thuoc\_tinh}”
* Bấm button thì form sẽ gửi request POST có chứa Todo lên địa chỉ /addTodo.

1. **POST /addTodo – thêm công việc vào list**

webController.java

/\* @ModelAttribute đánh dấu đối tượng Todo được gửi lên bởi Form Request\*/

@PostMapping("/addTodo")

public String addTodo(@ModelAttribute Todo todo) {

// Thêm đối tượng todo vào list

todoList.add(todo);

// Trả về trang thành công success.html

return "success";}

succes.html

<!DOCTYPE html>

<htmlxmlns:th="http://www.thymeleaf.org">

<head><metacharset="UTF-8"/>

<title>Loda To-do</title><metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><linkth:href="@{/css/main.css}"rel="stylesheet"/>

</head>

<body>

<h1>To-do</h1>

<h1>Thêm thành công!</h1>

<a th:href="@{/listTodo}"class="btn btn-primary">Xem danh sách công việc</a>

</body>

</html>

Output:

* Sau khi hoàn thiện xong các phần trên, chúng ta sẽ bổ sung index.html và App.java để khởi động server Web

App.java

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

Index.java

<!DOCTYPE html>

<htmlxmlns:th="http://www.thymeleaf.org"><head><metacharset="UTF-8"/><title>Loda To-do</title><metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/><!--css-->

<linkth:href="@{/css/bootstrap.css}"rel="stylesheet"/><linkth:href="@{/css/main.css}"rel="stylesheet"/><!--js-->

<scriptth:src="@{/js/bootstrap.js}"></script></head><body><h1>Todo App</h1><ath:href="@{/listTodo}"class="btn btn-primary">Xem danh sách</a><ath:href="@{/addTodo}"class="btn btn-success">+ Thêm công việc</a></body></html>

* Khi khởi động ứng dụng truy cập địa chỉ: <http://localhost:8080/>
* Nhấn nút xem danh sách để đi tới GET /listTodo
* Bấm thêm công việc để đi tới tỏng GET /addTodo
* Nhập tông tin và bấm add để gửi thông tin tới server POST /addTodo.
* Nếu thành công, server sẽ trả về thành công.
* Xem lại danh sách công việc bằng cách bấm vào xem danh sách công việc.
* Nếu danh sách quá nhiều, chúng ta có gới hạn bằng cách truyền lên param limit.
* Vd: <http://localhost:8080/listTodo?limit=2/>

XII, [Basic] Hướng dẫn Spring Boot JPA + MySQL

1, giới thiệu

* Để đi tiếp trong series Spring Boot này, tôi không thể bỏ qua một phần quan trọng đó là giao tiếp với Database.
* Vì nếu thiếu phần kết nối với Database nên chúng ta chưa thể hoàn thiện được trang web của mình, trong bìa này chúng ta sẽ tìm hiểu **Spring Boot JPA**

2, Spring Boot JPA

* **Spring Boot JPA** là một phần trong hệ sinh thái Spring Data, nó tạo ra một layer ở giữa tầng service và database, giúp chúng ta thao tác với database một cách dễ dàng hơn, tự động config và giảm thiểu code bừa bãi.
* **Spring Boot JAPA** đã wrapper Hibernate và tạo ra một interface mạnh mẽ. nếu như bạn gặp khó khăn khi làm việc với Hibernate thì đừng lo, bạn hãy để **Spring JPA** làm hộ.

Cài đặt:

* Để thêm **Spring JPA** vào project, bạn cần thêm dependency spring-boot-starter-data-jpa.
* Ngoài ra, để connect tới Mysql, chúng ta cần driver tương ứng, vì vậy phải bổ sung thêm cả dependency mysql-connertor-java vào pom.xml

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!--spring jpa-->

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- mysql connector -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Cấu trúc thư mục:

1. Tạo Table và dữ liệu

* Trước khi bắt đầu, chúng ta cần tạo ra dữ liệu trong Database. ở đây tôi chọn MySql.
* Dưới đây là SQL Script để tạo Database micro\_db. Chứa một table duy nhất là user.
* Khi chạy script này, nó sẽ tự động insert vào db 100 user

CREATE DATABASE micro\_db;

use micro\_db;

CREATE TABLE `user`

(

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`hp` int NULL DEFAULT NULL,

`stamina` int DEFAULT NULL,

`atk` int DEFAULT NULL,

`def` int DEFAULT NULL,

`agi` int DEFAULT NULL,

PRIMARY KEY (`id`)

);

DELIMITER $$

CREATE PROCEDURE generate\_data()

BEGIN

DECLARE i INT DEFAULT 0;

WHILE i < 100 DO

INSERT INTO `user` (`hp`,`stamina`,`atk`,`def`,`agi`) VALUES (i,i,i,i,i);

SET i = i + 1;

END WHILE;

END$$

DELIMITER ;

CALL generate\_data();

1. Tạo Model User

* Khi đã có dữ liệu trong Database. Chúng ta sẽ tạo một class trong java để mapping thông tin.
* Phần này chúng ta cần có một chút kiến thức về Hibernate. Nếu bạn chưa biết những annotation ở dưới đây để làm gì thì hãy tạm dừng và tìm hiểu Hibernate.

User.java

@Entity

@Table(name = "user")

@Data

public class User implements Serializable {

private static final long serialVersionUID = -297553281792804396L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) // tự gen dữ liệu

private Long id;

// Mapping thông tin biến với tên cột trong Database

@Column(name = "hp")

private int hp;

@Column(name = "stamina")

private int stamina;

// Nếu không đánh dấu @Column thì sẽ mapping tự động theo tên biến

private int atk;

private int def;

private int agi;

}

1. **Vấn đề của Hibernate truyền thống**

* Thông thường, khi bạn đã định nghĩa Entity tương ứng với Table trong DB thông qua Hibernate. Thì nhiệm vụ tiếp theo sẽ là tạo ra các class thao tác với DB.

Vd:

* Muốn query lấy tất cả User bằng Hibernate truyền thống sẽ như sau

// Giả sử đã có đối tượng session rồi

Session session = getSession();

try {

// Tất cả các lệnh hành động với DB thông qua Hibernate

// đều phải nằm trong 1 giao dịch (Transaction)

// Bắt đầu giao dịch

session.getTransaction().begin();

// Tạo một query

String sql = "Select u from " + User.class.getName() + " u ";

// Tạo đối tượng Query.

Query<User> query = session.createQuery(sql);

// Thực hiện truy vấn và lấy ra dữ liệu.

List<User> users = query.getResultList();

// In ra màn hình

for (User user : users) {

System.out.println(user);

}

// Commit dữ liệu và kết thúc session.

session.getTransaction().commit();

} catch (Exception e) {

e.printStackTrace();

// Rollback trong trường hợp có lỗi xẩy ra.

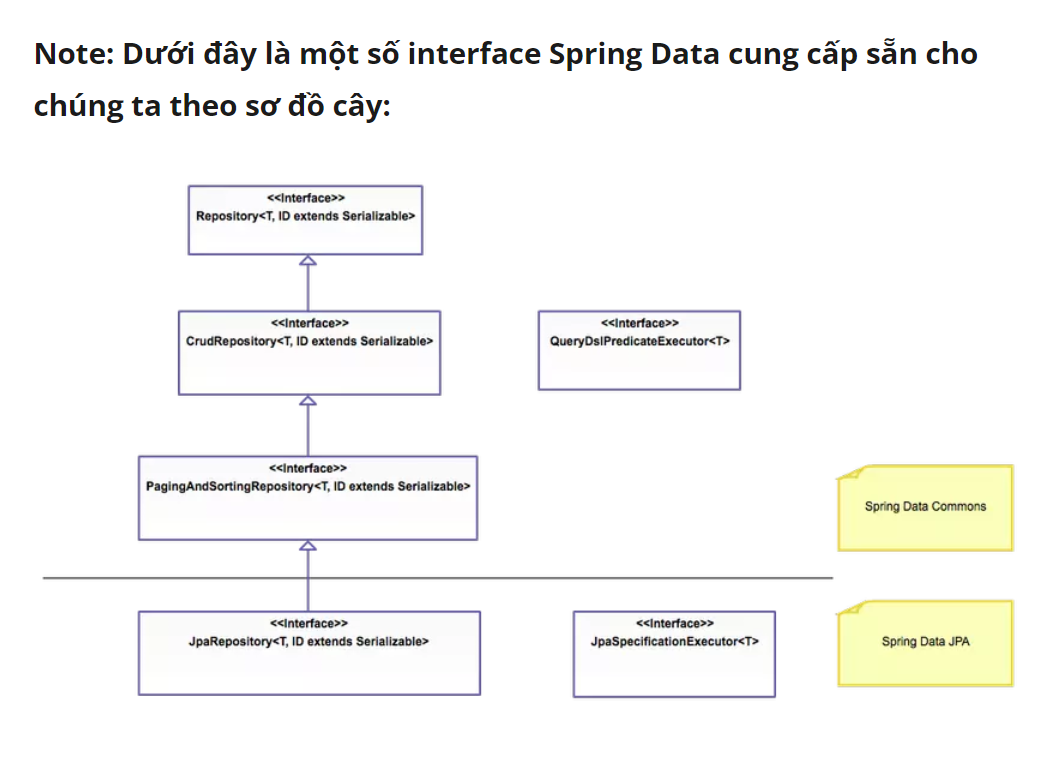
session.getTransaction().rollback();

}

* Mặc dù Hibernate đã làm rất tốt và giảm thiểu code cho việc thao tác với Database xuống rồi, nhưng nó vẫn chưa hẳn là dễ dàng =))
* Mục đích ban đầu của Hibernate là giúp người lập trình dễ sử dụng, tuy nhiên trên thực tế, nhiều người gặp khó khăn trong việc sử dụng Hibernate hơn là jdbc.
* Nắm được vấn đề này, Spring Data đã wrapper lên Hibernate một lớp nữa gọi là Spring JPA, giúp cho mọi người thao tác với DB của chúng ta rút ngắn xuống còn 1 dòng và tất nhiên là làm mờ Hibernate xuống đáng kể để tránh rắc rối cho người lập trình

1. **Curd với JPA**

* trong JPA cung cấp cho chúng ta 1 số interface để có thể thao tac với csdl như sau



* CurdRepository: save(), delete(), deleteById(), findById(), findAll(), v.v
* PagingAndSortingRepository: thêm phân trang
* JpaRepository: sử dụng cho mọi loại id, và có thể tự custom

* Để sử dụng Spring JPA, bạn cần sử dụng interface JpaRepository
* Yêu cầu của interface này đó là bạn phải cung cấp 2 thông tin:

1. Entity (Đối tượng tương ứng với Table trong DB)
2. Kiểu dữ liệu của khóa chính (primary key)

Vd:

* Tôi muốn lấy thông tin của bảng user thì làm như sau:

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

}

* Vậy thôi, @Repository đánh dấu UserRepository là một Bean và chịu trách nghiệm giao tiếp với DB.
* Spring Boot sẽ tự tìm thấy và khởi tọa ra đối tượng UserRepository trong Context. Việc tạo ra UserRepository hoàn toàn tự động và tự config, vì chúng ta đã kế thừa JpaRepository.
* Bây giờ, việc lấy ra toàn bộ user sẽ như sau:

@Autowired

UserRepository userRepository;

userRepository.findAll()

.forEach(System.out::println);

* Đơn giản và ngắn gọi hơn rất nhiều.
* Nếu bạn tìm kiếm thì sẽ thấy UserRepository có hàng chục method mà chúng ta không cần viết lại nữa. vì nó kế thừa JpaRepository

Application.properties

// file kết nối db

spring.datasource.url=jdbc:mysql://localhost:3306/micro\_db?useSSL=false

spring.datasource.username=root

spring.datasource.password=root

## Hibernate Properties

# The SQL dialect makes Hibernate generate better SQL for the chosen database

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

logging.level.org.hibernate = ERROR

* Spring JPA sẽ tự kết nối cho chúng ta, mà không cần thêm một đoạn code nào cả.

User.java

@Entity

@Table(name = "user")

@Data

public class User implements Serializable {

private static final long serialVersionUID = -297553281792804396L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

// Mapping thông tin biến với tên cột trong Database

@Column(name = "hp")

private int hp;

@Column(name = "stamina")

private int stamina;

// Nếu không đánh dấu @Column thì sẽ mapping tự động theo tên biến

private int atk;

private int def;

private int agi;

}

App.java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import lombok.RequiredArgsConstructor;

@SpringBootApplication

@RequiredArgsConstructor

public class App {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(App.class, args);

UserRepository userRepository = context.getBean(UserRepository.class);

// Lấy ra toàn bộ user trong db

userRepository.findAll()

.forEach(System.out::println);

// Lưu user xuống database

User user = userRepository.save(new User());

// Khi lưu xong, nó trả về User đã lưu kèm theo Id.

System.out.println("User vừa lưu có ID: " + user.getId());

Long userId = user.getId();

// Cập nhật user.

user.setAgi(100);

// Update user

// Lưu ý, lúc này đối tượng user đã có Id.

// Nên nó sẽ update vào đối tượng có Id này

// chứ không insert một bản ghi mới

userRepository.save(user);

// Query lấy ra user vừa xong để kiểm tra xem.

User user2 = userRepository.findById(userId).get();

System.out.println("User: " + user);

System.out.println("User2: " + user2);

// Xóa User khỏi DB

userRepository.delete(user);

// In ra kiểm tra xem userId còn tồn tại trong DB không

User user3 = userRepository.findById(userId).orElse(null);

System.out.println("User3: " + user2);

}

}

Output:

User vừa lưu có ID: 104

// sau khi update, cả 2 đối tượng user đều có giá trị agi mới

User: User(id=104, hp=0, stamina=0, atk=0, def=0, agi=100)

User2: User(id=104, hp=0, stamina=0, atk=0, def=0, agi=100)

// Sau khi xóa, user không còn tồn tại

User3: null