**EX04\_SPRING BOOT BASIC**

**BÁO CÁO THU HOẠCH**

1. **Tight coupled và Loosely coupled**

**Tight coupling** hay "liên kết ràng 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 sẽ dẫn tới ảnh hưởng tới toàn bộ các Class khác.

**Loosely-coupled** là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các Class với nhau.

Để đạt được Loosely-coupled, Java cung cấp Interface để giúp cho lớp có một quan hệ tổng quát với tập các lớp cùng nghiệp vụ, thay vì phải hard-code hay tạo ra nhiều ràng buộc với các lớp riêng khác nhau. Việc Loosely-coupled cũng được thể hiện bằng cách ta sử dụng một hàm nhưng với tham số đầu vào tổng quát được cấu hình, hay qua cấu hình cho phương thức khởi tạo, để nó có thể phục vụ cho nhiều đối tượng khác nhau mà không phải tạo nhiều hàm tương tự.

Ví dụ về Loosely-coupled

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 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);

}

(Tham khảo từ : <https://viblo.asia/p/khai-niem-tight-coupling-lien-ket-rang-buoc-va-cach-loosely-coupled-63vKjVERK2R> )

1. **Dependency Injection**

**Inversion of Control (IoC)**

IoC nhằm mục đích đơn giản hóa quá trình tạo đối tượng và liên kết giữa chúng, bằng cách tuân theo nguyên tắc: Không tạo đối tượng, chỉ mô tả cách chúng sẽ được tạo ra.

IoC framework sẽ có các thành phần có sẵn làm nhiệm vụ tạo, quản lý các đối tượng trong chương trình. IoC sẽ quản lý, phân tích các mối phụ thuộc, tạo các đối tượng theo thứ tự phù hợp nhất và liên kết chúng lại với nhau, theo cách lập trình viên mô tả.

Ví dụ

@Component

class VNEngine implements Engine {

...

}

@Component

class Car {

// Tìm object tương ứng với Engine và chèn (inject) vào đây

@Autowired

private Engine engine;

}

Mỗi class được đánh dấu @Component (cái này gọi là Annotation trong java) sẽ được IoC hiểu là một module:

* @Component là bảo IoC container tạo một object duy nhất (singleton)
* @Autowired là tìm module tương ứng (tạo từ trước) và inject vào đó.

Nhờ có IoC framework, việc tạo và liên kết các module dễ dàng hơn nhiều. Tuy nhiên, IoC chỉ nên áp dụng trên các module, object lớn, còn các object nhỏ, dùng tạm thời thì không nên dùng.

**Dependency injection**

DI là một dạng thực hiện của IoC, bằng cách tiêm (inject) module vào một module khác cần nó. Ví dụ module cấp cao Car cần có module Engine, thì dependency injection sẽ thực hiện theo các bước như sau:

* Tìm và tạo module tương ứng với Engine (class nào có implements Engine interface)
* Tạo tiếp module Car, do Car phụ thuộc vào Engine nên IoC tìm và lấy ra đối tượng Engine đã tạo trước đó và inject vào bên trong Car.

Nếu khi tạo module nào đó, mà module đó cần một module khác phụ thuộc, thì IoC sẽ tìm trong IoC container xem có không, nếu có thì inject vào, nếu chưa thì tạo mới, bỏ vào container và inject vào. Việc inject tự động các dependency (module) như thế được gọi là Dependency injection, điều này giúp tiết kiệm bộ nhớ và quản lý cũng dễ hơn.

**2 loại injection**

* Constructor-based injection: Dùng inject các module bắt buộc. Các module được inject nằm trong constructor, và được gán lần lượt vào các field.
* Setter-based injection: Dùng inject các module tùy chọn. Mỗi module sẽ được inject thông qua setter, nằm ở tham số và cũng gán cho field nào đó.

Ví dụ:

@Component

class Car {

// Bắt buộc, vì xe thì phải có động cơ

private Engine engine;

// Tùy chọn, vì xe có thể không có người chủ

private Human owner;

// Do engine bắt buộc, nên dùng constructor based injection

// Constructor based có thể inject nhiều dependency cùng lúc

public Car(Engine engine) {

this.engine = engine;

}

// Do owner là tùy chọn, nên dùng setter based injection

// Setter based chỉ inject một dependency mỗi setter

public void setOwner(Human owner) {

this.owner = owner;

}

}

(Tham khảo từ : <https://viblo.asia/p/dependency-injection-ap-dung-vao-spring-boot-nhu-the-nao-phan-2-3P0lP1wG5ox> )

**Áp dụng vào Spring Boot**

Spring là một framework được xây dựng dựa trên nguyên lý Dependency injection. Bản thân Spring có chứa IoC container, có nhiệm vụ tạo và quản lý các module:

* IoC container của Spring gọi là **Application context**
* Các module chứa trong IoC container được Spring gọi là các **Bean**

Spring Boot sử dụng các **annotation** dạng như @Component để đánh dấu lên class, chỉ ra rằng class đó cần tạo một module. Ngoài @Component, còn có các annotation khác như @Repository, @Controlller, @Service,... cũng được đánh dấu là module.

Khi ứng dụng Spring Boot chạy, thì IoC container sẽ thực hiện quá trình như sau:

* Quét tìm (scan) các class được đánh dấu là Bean, và tạo một object singleton, bỏ vào IoC container
* Khi có một Bean phụ thuộc vào Bean khác, thì IoC sẽ tìm trong container, nếu chưa có thì tạo, nếu đã có thì lấy ra và inject vào bean cần nó

1. **Spring Profile**

**Spring Profiles** là một core feature trong Spring Framework, cho phép chúng ta cấu hình ứng dụng, active/deactive Bean tùy theo môi trường.

Ta thường cấu hình trong file application.yml hoặc application.properties

VD file application.yml:

loda:

email: loda.namnh@gmail.com

googleAnalyticsId: U-xxxxx

Để sử dụng trong 1 lớp nào đó, ta khai báo nó là @Component, chỉ định @Value cho thuộc tính (ví dụ @Value(“${loda.email}”) hoặc đặt trùng tên thì Spring sẽ tự động binding

@Component // Là 1 spring bean

// @PropertySource("classpath:loda.yml") // Đánh dấu để lấy config từ trong file loda.yml

@ConfigurationProperties(prefix = "loda") // Chỉ lấy các config có tiền tố là "loda"

public class LodaAppProperties {

private String email;

private String googleAnalyticsId;

// standard getters and setters

}

Để lấy thông tin Config, ta dùng annotation @Autowired và kích hoạt với @EnableConfigurationProperties

@SpringBootApplication

@EnableConfigurationProperties

public class App implements CommandLineRunner {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

@Autowired LodaAppProperties lodaAppProperties;

@Override

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

System.out.println("Global variable:");

System.out.println("\t Email: "+lodaAppProperties.getEmail());

System.out.println("\t GA ID: "+lodaAppProperties.getGoogleAnalyticsId());

}

}

Kết quả:

Global variable:

Email: loda.namnh@gmail.com

GA ID: U-xxxxx

**Nhiều Profile khác nhau**

Ví dụ ta có hệ thống Profile như sau:

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

Chi tiết:

* *application.yml*

#application.yml

---

spring.profiles: local

spring.profiles.include: common, local

---

spring.profiles: aws

spring.profiles.include: common, aws

---

* *application-aws.yml*

spring:

datasource:

username: xxx

password: xxx

url: jdbc:mysql://10.127.24.12:2030/news?useSSL=false&characterEncoding=UTF-8

* *application-local.yml*

spring:

datasource:

username: root

password:

url: jdbc:mysql://localhost:3306/news?useSSL=false&characterEncoding=UTF-8

logging:

level:

org:

hibernate:

SQL: debug

* *application-common.yml*

spring:

jpa:

properties:

hibernate:

jdbc:

batch\_size: 50

batch\_versioned\_data: true

hibernate:

ddl-auto: none

File **application.yml** khai báo 2 môi trường là **local** và **aws**. Tại mỗi môi trường sẽ include (bao gồm) các file config như trên. Khi kích hoạt **aws** chẳng hạn, Spring sẽ load tất cả config có trong **application-common.yml** và **application-aws.yml.**

**Kích hoạt Config**

4 cách:

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

spring.profiles.active=aws

* #2: Active trong code, trước khi chạy chương trình (not recommend)

@Configuration

public class ApplicationInitializer

implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletContext) throws ServletException {

servletContext.setInitParameter(

"spring.profiles.active", "aws");

}

}

Hoặc

@Autowired

private ConfigurableEnvironment env;

...

env.setActiveProfiles("aws");

hoặc

SpringApplication application = new SpringApplication(SpringBootProfilesApplication.class);

ConfigurableEnvironment environment = new StandardEnvironment();

environment.setActiveProfiles("aws");

application.setEnvironment(environment);

application.run(args);

* Sử dụng JVM System Parameter (nên dùng)

-Dspring.profiles.active=aws

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

export SPRING\_PROFILES\_ACTIVE=aws

**Sử dụng @Profile**

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

// Bean này Spring chỉ khởi tạo và quản lý khi môi trường là `local`

@Component

@Profile("local")

public class LocalDatasourceConfig

Cho phép dùng toán tử logic

// Bean này Spring chỉ khởi tạo và quản lý khi môi trường là những môi trường không phải là `local`

@Component

@Profile("!local")

public class LocalDatasourceConfig

@Component: Đánh dấu một class là bean và để cho BeanContainer tạo một đối tượng của nó để quản lý, khi một class nào đó cần thì có thể inject cái class đã tạo này

@Autowired: Chỉ ra rằng hãy inject cái class trong container đi. Autowired chỉ hoạt động với Bean đầu tiên được beanContainer tạo. Các đối tượng tự khởi tạo sẽ không autoWired

Profile

@EnableConfigurationProperties: Cho phép sử dụng ConfigurationProperties

@ConfigurationProperties: class này sử dụng file config cho các thuộc tính trùng tên của nó (tương tự chỉ áp dụng cho Bean được tạo đầu tiên)

@Value: dùng đối tượng trong file config (tương tự chỉ áp dụng cho Bean được tạo đầu tiên)