# **BÁO CÁO NHỮNG KIẾN THỨC TÌM HIỂU ĐƯỢC**

1. **Tight coupled và cách loosely coupled**

* **Giới thiệu**

**Tight coupled** 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.

* **Ví dụ:**

Ví dụ 1

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 {

private BubbleSortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

public VeryComplexService(){

}

public void complexBusiness(int array[]){

bubbleSortAlgorithm.sort(array);

// TODO: more 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 cả 2 Class trên. 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 implement này, nó là liên kết rất chặt với nhau.

Ví dụ 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 **SortAlgorithm**. Với cách này thì mỗi quan hệ giảm bớt sự liên kết, nhưng nó không thay đổi được việc thuật toán vẫn đang là **BubbleSortAlgorithm**.

Ví dụ 3:

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);

}

}

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 làm phổ biển nhất. Mối liên hệ giữa 2 Class đã "lỏng lẻo" hơn trước rất nhiều. **VeryComplexService** sẽ không quan tâm tới việc thuật toán sắp xép 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.

1. **Dependency injection**

**Dependency Injection cụ thể là gì?**

Theo Wikipedia: "Trong kỹ thuật phần mềm, dependency injection là một kỹ thuật theo đó một đối tượng (hoặc static method) cung cấp các phụ thuộc của đối tượng khác. Một phụ thuộc là một đối tượng có thể được sử dụng (service)."

Nhưng nó vẫn khá khó hiểu, vậy hãy cũng làm rõ nó.

Đầu tiên, dependency hay dependent nghĩa là phụ thuộc vào hỗ trợ của một cái gì, việc gì đó. Ví dụ như nếu chúng ta phụ thuộc quá nhiều vào smartphone, thì có thể hiểu là chúng ta đã dependent lên smartphone, chúng ta phụ thuộc vào robot, đấy chính là chúng ta đã dependent lên robot.

Trước khi nói về dependency injection, hãy hiểu xem dependency trong lập trình nghĩa là gì trước.

Khi mà class A sử dụng một số chức năng của class class B, thì có thể nói là class A có quan hệ phụ thuộc với class B.

Trong java, trước khi ta có thể sử dụng method của class khác, ta phải khởi tạo một object của class đấy (hay A cần phải tạo 1 thực thể của B). Vậy ta có thể hiểu, việc chuyển giao nhiệm vụ khởi tạo object đó cho một ai khác và trực tiếp sử dụng các dependency đó được gọi là dependency injection.

Hay nói một cách chính xác và khách quan hơn là:

Dependency injection (DI) là một kỹ thuật lập trình giúp tách một class độc lập với các biến phụ thuộc. Với lập trình hướng đối tượng, chúng ta hầu như luôn phải làm việc với rất nhiều class trong một chương trình. Các class được liên kết với nhau theo một mối quan hệ nào đó. Dependency là một loại quan hệ giữa 2 class mà trong đó một class hoạt động độc lập và class còn lại phụ thuộc bởi class kia.

**Thông thường, chúng ta chỉ thường gặp ba loại Dependency Injection sau:**

* Constructor injection: Các dependency (biến phụ thuộc) được cung cấp thông qua constructor (hàm tạo lớp).
* Setter injection: Các dependency (biến phụ thuộc) sẽ được truyền vào 1 class thông qua các setter method (hàm setter).
* Interface injection: Dependency sẽ cung cấp một Interface, trong đó có chứa hàm có tên là Inject. Các client phải triển khai một Interface mà có một setter method dành cho việc nhận dependency và truyền nó vào class thông qua việc gọi hàm Inject của Interface đó.

**Nhiệm vụ của Dependency Injection là:**

1. Tạo ra các object.
2. Biết được class nào cần những object đấy.
3. Cung cấp cho những class đó những object chúng cần.

**Lợi ích và bất cập khi dùng Dependency Injection**

Lợi ích khi dùng Dependency Injection:

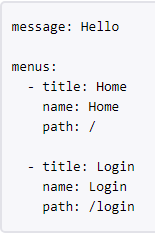
* Dễ test và viết Unit Test: Dễ hiểu là khi ta có thể Inject các dependency vào trong một class thì ta cũng dễ dàng “Inject” các mock object vào class (được test) đó.
* Dễ dàng thấy quan hệ giữa các object: Dependency Injection sẽ inject các object phụ thuộc vào các interface thành phần của object bị phụ thuộc nên ta dễ dàng thấy được các dependency của một object.
* Dễ dàng hơn trong việc mở rộng các ứng dụng hay tính năng.
* Giảm sự kết dính giữa các thành phần, tránh ảnh hưởng quá nhiều khi có thay đổi nào đó.

**Bất lợi của Dependency Injection:**

* Nó khá là phức tạp để học, và nếu dùng quá đà thì có thể dẫn tới một số vấn đề khác.
* Rất nhiều các lỗi ở compile time có thể bị đẩy sang runtime, dẫn đến đôi khi sẽ khó debug. Vì sử dụng các Interface nên có thể gặp khó khăn khi ta debug source code vì không biết implement nào thực sự được truyền vào.
* Có thể làm ảnh hưởng tới chức năng auto-complete hay find references của một số IDE. Cụ thể vì Dependency Injection ẩn các dependency nên một số lỗi chỉ xảy ra khi chạy chương trình thay vì có thể phát hiện khi biên dịch chương trình.
* Khó khăn lớn nhất là khi người mới vào làm bằng DI sẽ không hiểu rõ ràng tư tưởng, khiến quá trình làm DI vẫn bị nhập nhằng và các injector bị ràng buộc mà không thoát hẳn ra theo tư tưởng của DI.

1. **File cấu hình application.properties, application.yml**
2. **YAML(yml)**

YAML (YAML Ain’t Markup Language) là một lại ngôn ngữ tương tự JSON được thiết kế **với** mục đích để người và máy cùng đọc được.

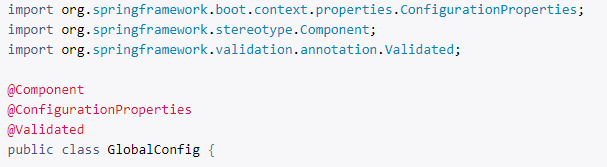


YAML file thường được dùng để làm các file cấu hình (config), lưu giá trị… Nhìn chung mục đích của file YAML giống với file .properties

File YAML trong Spring Boot. Trong Spring Boot file YAML (.yml) được dùng để làm file config (tương đương với file .properties).

Ta có thể dùng file YAML hoặc file Properties hoặc kết hợp cả 2. (Xem lại: Đọc file .properties với Spring Boot)

Việc đọc file YAML trong Spring Boot giống với việc đọc file Properties. Chúng ta cũng sử dụng các annotation @Value, @ConfigurationProperties

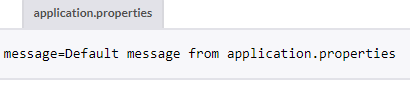


1. **File .Properties**

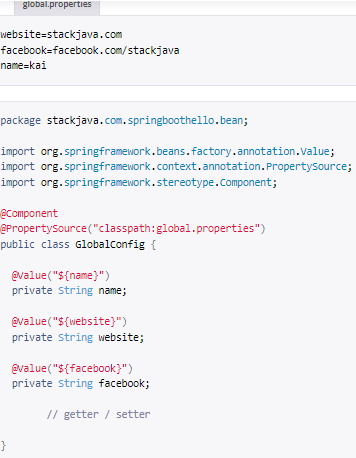
**Cách sử dụng căn bản**

**Sử dụng annotation @Value**

Cách đơn giản nhất là sử dụng annotation @Value để inject giá trị trong file application.properites vào thuộc tính của các Spring Bean.

**Sử dụng annotation @PropertySource để đọc những file .properites tùy chọn.**



**Sử dụng annotation @EnableAutoConfiguration**

Annotation @EnableAutoConfiguration sẽ tự động mapping thuộc tính của bean với các giá trị trong file .properties

Ví dụ:

package stackjava.com.springboothello.bean;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.PropertySource;

import org.springframework.stereotype.Component;

@Component

@ConfigurationProperties

@PropertySource("classpath:global.properties")

public class GlobalConfig {

private String name;

private String website;

private String facebook;

// getter / setter

}

## **Đọc các file properties phức tạp**

Ví dụ:

**menu.properties**

#App

app.menus[0].title=Home

app.menus[0].name=Home

app.menus[0].path=/home

app.menus[1].title=Login

app.menus[1].name=Login

app.menus[1].path=/login

Hoặc với file yaml

**menu.yml**

app:

menus:

- title: Home

name: Home

path: /

- title: Login

name: Login

path: /login

**\*Lưu ý:**annotation @ConfigurationProperties hỗ trợ cả file.properties và file.yml.

package stackjava.com.springboothello.bean;

import java.util.ArrayList;

import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.PropertySource;

import org.springframework.stereotype.Component;

@Component

@ConfigurationProperties("app") // prefix app, find app.\* values

@PropertySource("classpath:menu.properties")

public class MenuConfig {

private List<Menu> menus = new ArrayList<>();

public List<Menu> getMenus() {

return menus;

}

public void setMenus(List<Menu> menus) {

this.menus = menus;

}

public static class Menu {

private String name;

private String path;

private String title;

// getter - setter

@Override

public String toString() {

return "Menu{" + "name='" + name + '\'' + ", path='" + path + '\'' + ", title='"+ title + '\'' + '}';

}

}

**Sử dụng Environment**

Bạn chỉ cần inject Environment vào các component là có thể lấy các message trong file .properties. Mặc định Environment lấy message từ trong file application.properties, nếu bạn muốn nó đọc từ cả các file khác thì dùng kèm với @PropertySource

1. **Chạy nhiều môi trường với Spring Profile**

Spring Profiles là tính năng giúp lựa chọn các cấu hình khác nhau tùy theo môi trường phát triển.

Trong một dự án việc tạo ra các môi trường là rất cần thiết cho việc phát triển, thử nghiệm và release sản phẩm. Vì một số config và đặc biệt một số phần trong code cũng có thể thay đổi tùy vào môi trường, sớm nhận ra những khó khăn trong việc config khi xây dựng ứng dụng, Spring đã cho ra đời Spring Profiles để giải quyết vấn đề này.



**Validate dữ liệu từ file properties**

@ConfigurationProperties Hỗ trợ validate dữ liệu với javax.validation

Để validate dữ liệu ta sử dụng annotation @Validated ở đầu class và các annotation validate ở mỗi field ví dụ:

**GlobalConfig.java**

package stackjava.com.springboothello.bean;

import javax.validation.constraints.NotEmpty;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.PropertySource;

import org.springframework.stereotype.Component;

import org.springframework.validation.annotation.Validated;

@Component

@ConfigurationProperties

@PropertySource("classpath:global.properties")

@Validated

public class GlobalConfig {

@NotEmpty

private String name;

private String website;

private String facebook;

// getter - setter

}

1. **Tạo file cấu hình**

Mặc định Spring boot sẽ lấy thông tin cấu hình từ các file .properties or .yaml

Để cấu hình riêng cho từng môi trường trong ví dụ này mình sẽ tạo ra 3 file config tương ứng với 3 môi trường đó là:

application.{properties|yaml} (default)

application-dev.{properties|yaml}

application-staging.{properties|yaml}

application-prod.{properties|yaml}

1. **Với file cấu hình properties, yaml**

Ví dụ: Config datasource cho từng môi trường

Với môi trường develop application-dev.properties

cafeit.datasource=jdbc:mysql://localhost:3306/database-dev

Với môi trường staging application-staging.properties

cafeit.datasource=jdbc:mysql://10.10.10.11:3306/database-staging

Với môi trường production application-prod.properties

cafeit.datasource=jdbc:mysql://10.10.10.10:3306/database-prod

### **Với bean và component**

Bất kỳ các Spring bean, component (@Component, @Configuration, @Bean) đều có thể được đánh dấu @Profile để giới hạn khi tải. Ví dụ: Config datasource cho từng môi trường Với môi trường develop:

@Configuration

@Profile(**"dev"**)

**public** **class** DevelopConfig {

@Value(**"${cafeit.datasource}"**)

**private** String datasource;

@Bean

**public** DataSource dataSource() {

**return** **new** DataSource(datasource);

}

}

Với môi trường staging:

@Configuration

@Profile(**"staging"**)

**public** **class** StagingConfig {

@Value(**"${cafeit.datasource}"**)

**private** String datasource;

@Bean

**public** DataSource dataSource() {

**return** **new** DataSource(datasource);

}

}

Với môi trường Production:

@Configuration

@Profile(**"prod"**)

**public** **class** ProductionConfig {

@Value(**"${cafeit.datasource}"**)

**private** String datasource;

@Bean

**public** DataSource dataSource() {

**return** **new** DataSource(datasource);

}

}

Annotation @Value là annotation khai báo ở mức variable có chức năng load một giá trị từ file properties|yaml lên một biến có kiểu dữ liệu tương ứng.

Annotation @Configuration là annotation khai báo ở mức class cho chỉ ra cho Spring biết đây là nơi định nghĩa các bean.

Annotation @Profile là annotation khai báo ở mức class và method có chức năng khai báo cho Spring biết class hoặc bean này sẽ được chạy ở môi trường(profile) nào.

1. **Kích hoạt môi trường**

Để active môi trường(profile) sẽ có các cách sau:

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

spring.profiles.active=**dev**

Cách 2: Active trong code, trước khi chạy chương trình.

[spring.profiles.active = dev

@Configuration public class ApplicationInitializer triển khai WebApplicationIni]

@Configuration

**public** **class** ApplicationInitializer

**implements** WebApplicationIni

tializer {

@Override

**public** **void** onStartup(ServletContext servletContext) **throws** ServletException {

servletContext.setInitParameter(

**"spring.profiles.active"**, **"dev"**);

}

}

Hoặc

@Autowired

**private** ConfigurableEnvironment env;

...

env.setActiveProfiles(**"dev"**);

Cách 3: Sử dụng JVM System Parameter (nên dùng)

-Dspring.profiles.active=dev

Cách 4: Environment Variable (Unix) (nên dùng)

export SPRING\_PROFILES\_ACTIVE=dev

**Run**

spring.profiles.active=[tên môi trường]

@SpringBootApplication

**public** **class** SpringProfileApplication **implements** CommandLineRunner {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SpringProfileApplication.class, args);

}

@Autowired

**private** DataSource dataSource;

@Override

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

System.out.println(**"Data source: "** + dataSource.getUrl());

}

}