**BÁO CÁO**

1. **Tigh-coupling và loose-coupling**

Trước khi tìm hiểu về Tight-coupling và loose-coupling ta tìm hiểu POJI và POJO là gì.

1. **POJI**

La viết tắt của Plain Old Java Interface. Một POJI là một interface Java thuần, nó không extend interface của bất kì framework hay technology nào.

Ví dụ:

// POJI

public interface MyInterface1 {

public void Method1();

public Integer Method2();

}

// MyInterface1 là POJI

public interface MyInterface2 extends Serializable {

public void Method1();

public Integer Method2();

}

// MyInterface2 là POJI vì Serializable là một core interface (tức nó là một POJI)

public interface MyInterface3 extends MyInterface1 {

// Some themods

}

// MyInterface3 là POJI vì MyInterface1 là user interface.

// --------------------------------------------------

// Not POJI

public interface MyServlet extends javax.servlet.Servlet {

// Some themods

}

public interface MyInterface4 extends MyServlet {

// Some themods

}

// MyServlet và MyInterface4 không phải POJI vì Servlet không phải là POJI (là interface của framework Servlet)

Public interface MyInterface5 extends java.rmi.Remote {

// Some themods

}

// MyInterface5 không phải POJI vì Remote là interface của RMI technology.

1. **POJO**

Là viết tắt của Plain Old Java Object. Một POJO là một class Java thuần, tức là nó không extend/implement bất kỳ class/interface của framework hay một technology nào.

Ví dụ: // POJO

public class MyClass1 {

// Some methods

}

// MyClass1 là một POJO

public class MyClass2 implements Serializable {

// Some methods

}

// MyClass2 là một POJO vì Serializable là một POJI

public class MyClass3 extends MyClass1 {

// Some methods

}

// MyClass3 là một POJO vì MyClass1 là một POJO

public class MyClass implements MyInterface1 {

// Some methods

}

// MyClass là một POJO vì MyInterface1 là một POJI.

// Ta gọi là POJI/POJO model.

// --------------------------------------------------

// Not POJO

public class MyClass4 implements javax.servlet.Servlet {

// Some methods

}

// MyClass4 không là một POJO vì Servlet không phải một POJI

public class MyClass5 extends MyClass4 {

// Some methods

}

// MyClass5 không là một POJO vì MyClass4 không phải một POJO

**Tight-coupling và Loose-coupling**

1. **Tight-coupling**

Lập trình OOP nói chúng và Java nói riêng thì việc Class A có logic gọi logic của Class B là hết sức thông thường. Tại Class A sẽ khởi tạo 1 đối tượng của Class B để sử dụng logic của Class B. Khi đó ta nói tồn tại tight-coupling giữa Clas A và Class B.

public class Travel {

Car car = new Car();

void startJourney() {

car.travel();

}

}

public class Car {

void move() {

System.out.prinln("Move by Car");

}

}

public class Train {

void move() {

System.out.prinln("Move by Train");

}

}

Ở ví dụ trên ta nói class Travel dependent class Car, hay class Car dependency của class Travel. Trong trường hợp này class Travel tight coupled (phụ thuộc/ghép đôi một cách chặt chẽ) với class Car. Nghĩa là khi clas Cả thay đổi method move() thì clas Travel cũng phải thay đổi theo. Hay khi startJourney() muốn đổi từ Car sang Train thì ta phải viết lại Travel như sau.

public clas Travel {

Train train = new Train();

void startJourney() {

train.travel();

}

}

1. **Loose-coupling**

Để loại bỏ tight coupled giữa clas Travel và class Car ta phải tổ chức các class theo mô hình POJI/POJO như sau.

public interface Vehicle {

void move();

}

public class Car implements Vehicle {

@Override

void move() {

System.out.prinln("Move by Car");

}

}

public class Train implements Vehicle {

@Override

void move() {

System.out.prinln("Move by Train");

}

}

public clas Travel {

Vehicle v = new Car();

void setV(Vehicle v) {

this.v = v;

}

void startJourney() {

v.move();

}

}

Với cách tổ chức theo mô hình POJI/POJO như trên thì class Travel không còn bị phụ thuộc chặt chẽ vào class Cả hay class Train nữa. Ta nói class Travle và class Car là loose-coupling.

1. **Dependency injection**

Dependency injection là một kĩ thuật trong đó một object (hoặc một static method) cung cấp các dependencies của một object khác. Một dependency là một object mà có thể sử dụng (một service). Tuy nhiên định nghĩa trên vẫn khá là khó hiểu, vậy nên hãy cùng tìm hiểu để hiểu rõ hơn về nó nào.

### Về cơ bản có 3 loại dependency injection:

1. **Constructor injection**: các dependency đc cung cấp thông qua constructor của class.
2. **Setter injection**: client tạo ra một setter method để các class khác có thể sử dụng chúng để cấp dependency.
3. **Interface injection**: dependency sẽ cung cấp một hàm injector để inject nó vào bất kì client nào đc truyền vào. Các client phải implement một interface mà có một setter method dành cho việc nhận dependency.

Vậy trách nhiệm 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.

Bằng cách này, nếu trong tương lai object đó có sự thay đổi thì dependency injection có nhiệm vụ cấp lại những object cần thiết cho class.

### **Lợi ích của dependency injection.**

1. Giúp viết Unit test dễ dàng hơn.
2. Giảm thiểu đc boilerplate code vì việc khởi tạo dependency đc làm bởi một component khác.
3. Mở dụng dự án dễ dàng hơn.
4. Giúp ích trong việc liên kết lỏng (loose coupling) giữa các thành phần trong dự án.

### **Bất lợi khi dùng dependency injection.**

1. Nó khá là phức tạp để học, và nếu dùng quá đà thì có thể dẫn tới một số vấn đề khác.
2. Rất nhiều các lỗi ở compile time có thể bị đẩy sang runtime.
3. Có thể làm ảnh hưởng tới chức năng auto-complete hay Find references của một số IDE
4. **Các anotation thường dùng**

#### **1. @Configuration**

Được sử dụng để chỉ ra rằng class khai báo sử dụng Annotation @Configuration sẽ khai báo một hoặc nhiều @Bean method trong class đó. Những class khai báo với @Configuration sẽ được Spring Container quản lý và tạo Bean trong lúc chương trình đang chạy. Thông thường các bean cấu hình cho dự án tôi để trong này. Ví dụ cấu hình themeleaf, đa ngôn ngữ, và nhiều cấu hình khác cho ứng dụng.

#### **2. @Bean**

Method (phương thức) sử dụng @Bean ở phía trên sản xuất ra đối tượng bean và được quản lý bởi Spring Container. Bean Annotation có thể sử dụng với các tham số như name, initMethod hoặc destroyMethod.

#### **4. @ComponentScan**

Chúng ta sử dụng @ComponentScan để thông báo có Spring Container biết phải vào package nào trong dự án để quét các Annotation và tạo Bean.

#### **5. @Component**

Khi một class được đánh dấu là Component thì sẽ được tạo thành 1 Bean. Khi Spring Start thì nó quét qua các Annotation có đánh dấu là @Component thì nó sẽ tạo Bean cho class đó. Ví dụ ta có class Contact và ta đánh dấu nó là @Component thì Spring khi đọc qua class này nó sẽ tạo 1 Bean có tên là Contact trong Container của nó. Nếu có class nào dùng thì nó sẽ nhúng Bean này vào. Dùng @Component là để tạo ra một bean.

#### **6. @PropertySource và @Value**

Trong Spring, chúng ta sử dụng @PropertySource để cho Spring biết tìm các file properties cấu hình cho hệ thống ở đâu đồng thời sử dụng @Value để lấy các giá trị trong file properties.

#### **7. @Service**

Nếu một class được đánh dấu là @Service thì nó là kiểu đặc biệt của @Component. Nó được dùng để xử lý các nghiệp vụ của ứng dụng. Ví dụ như kế toán thì có nghiệp vụ là kiểm tra chi, quản lý thu. Lớp BookServiceImpl dưới đây được đánh dấu là @Service thì nó sẽ phụ trách xử lý các vấn đề liên quan đến nghiệp vụ.

#### **8. @Repository**

Nếu một class được đánh dấu là @Repository thì nó là kiểu đặc biệt của @Component . Nó được sử dụng để nói Bean này dùng để truy cập và thao tác xuống cơ sở dữ liệu. Class BookDaoImpl được đánh dấu với @Repository nghĩa là lớp này có nhiệm vụ thực hiện các câu lệnh truy vấn xuống cơ sở dữ liệu.

#### **9. @Autowire**

Tự động nhúng các Bean được Spring Container sinh ra vào Class có khai báo @Autowire. Khi Spring nó sẽ tìm kiếm bean có tên là BookDao trong container của nó, sau đó nhúng (hoặc tiêm) vào lớp BookServiceImple. Đây chính là cơ chế DI (Depedency Injection). Khi Spring bắt đầu chạy nó sẽ quét qua các lớp có sử dụng Annotation để tạo Bean đồng thời nó cũng quýt bên trong các Bean xem có khai báo @Autowire không nếu có nó sẽ tìm kiếm Bean tương ứng mà nó quản lý và nhúng vào.

#### **10. @Scope**

Khi bean được tạo ra thì nó có nhiều scope khác nhau. Scope ở đây là phạm vi Bean được sinh và và bị phá huỷ dưới sự quản lý của Spring Container. Khi Bean được sinh ra nó có 5 scope (phạm vi được sử dụng):

* Singleton: đây là scope mặc định của 1 bean khi được sinh ra. Nếu ta không khai báo scope cụ thể thì Bean sẽ lấy Singleton scope. Singleton Bean có nghĩa là Bean chỉ tạo ra 1 lần và được sử dụng trong Container . Chỉ duy nhất 1 Bean tồn tại trong container
* Prototype: ngược lại với singleton ta muốn có nhiều Bean (đối tượng) thì ta sử dụng scope prototype
* Request: Bean được sinh ra thông qua các request http (yêu cầu) từ người dùng. Chỉ được dùng trong các ứng dụng web
* Session: Bean được sinh ra thông qua các http session
* Global-session: Bean được sinh ra thông qua các request http (yêu cầu) từ người dùng. Chỉ được dùng trong các ứng dụng web

#### **11. @Valid**

Dùng để kiểm tra dữ liệu có đúng như mình mong muốn hay không. Ví dụ dưới đây mình mong muốn name là không được rỗng, author không được rỗng. Nếu dữ liệu bị rỗng thì @Validate sẽ bắt lỗi.

#### **12. @Controller**

Một class được đánh dấu là Controller thì để khai báo Class đó là một Controller và có nhiệm vụ Mapping Request trên URL vào các method tương ứng trong Controller. Ví dụ dưới đây tôi khai báo Class HomeController là một Controller. Khi người dùng gõ vào http://localhost:8080/ thì sẽ được xử lý bởi Class HomeController. Như vậy nhiệm vụ của Controller là điều hướng các request (yêu cầu) người dùng vào method xử lý.

#### **13. @RequestMapping**

Có nhiệm vụ ánh xạ các request (yêu cầu) người dùng vào method tương ứng trong Controller.

#### **14. @PathVariable**

PathVariable được sử dụng để xử lý những URL động, có một hoặc nhiều parameter trong URL.

#### **15. @RequestParam**

Chúng ta sử dụng @RequestParam để bắt các giá trị các tham số mà người dùng truyền vào trên URL theo định dạng key và value.

#### **16. @ModelAttribute**

Một trong những Annotation quan trọng trong Spring đó là @ModelAttribute. Chúng ta sử dụng ModelAttribute như một cầu nối giữa Controller và View. Từ Controller, chúng ta truyền các dữ liệu qua cho View thông qua ModelAttribute. Từ View, chúng ta sẽ sử dụng Themeleaf để đọc các dữ liệu từ model và hiển thị ra cho người dùng.

Tầng View, chúng ta sử dụng model để lấy các giá trị từ người dùng và gắn vào thuộc tính modelAttribute.

#### **17. @RequestBody**

@RequestBody được sử dụng để lấy các giá trị mà người dùng gửi lên Server mà các giá trị đó được chứa trong phần thân (body) của request.

#### **18. @ResponseBody**

Tôi sử dụng @ResponseBody để nói cho controller biết rằng ta sẽ trả về một đối tượng Object kiểu JSON cho client chứ tôi không render ra một trang view.

#### **19. @RequestHeader và @ResponseHeader**

@RequestHeader được sử dụng khi ta muốn lấy dữ liệu được truyền bằng Header của một request (yêu cầu từ client).

#### **20. @SessionAttribute**

Chúng ta sử dụng @SessionAttribute để lưu trữ các giá trị trong một phiên làm việc. Giống như mình làm một ứng dụng shopping cart . Khi người dùng chọn 1 sản phẩm thì mình dùng session mình lưu lại. Khi khách hàng thanh toán giỏ hàng thì ta lấy hết tất cả các mặt hàng chứa trong session ra và tính toán.