Lê Đức Quý

# Báo cáo

1. **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.
2. **Loosely-coupled** là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các Class với nhau.

|  |
| --- |
| 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);  } |

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 (DI)** là một design pattern, một kỹ thuật cho phép xóa bỏ sự phụ thuộc giữa các module, làm cho ứng dụng dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.

**Các dạng Dependency Injection:**

* Constructor Injection: Các dependency sẽ được container truyền vào (inject vào) 1 class thông qua constructor của class đó. Đây là cách thông dụng nhất.
* Setter Injection: Các dependency sẽ được truyền vào 1 class thông qua các hàm Setter.
* Fields/ properties: Các dependency sẽ được truyền vào 1 class một cách trực tiếp vào các field.
* Interface Injection: Class cần inject sẽ implement 1 interface. Interface này chứa 1 hàm tên Inject. Container sẽ injection dependency vào 1 class thông qua việc gọi hàm Inject của interface đó. Đây là cách rườm rà và cũng ít được sử dụng.
* Service Locator: nó hoạt động như một mapper, cho phép thay đổi code tại thời điểm run-time mà không cần biên dịch lại ứng dụng hoặc phải khởi động lại

**Ưu điểm và khuyết điểm của Dependency Injection**

Ưu điểm:

* Reduced dependencies: giảm sự kết dính giữa các module.
* Reusable: code dễ bảo trì, dễ tái sử dụng, thay thế module. Giảm boiler-plate code do việc tạo các biến phụ thuộc đã được injector thực hiện.
* Testable: rất dễ test và viết Unit Test.
* Readable: dễ dàng thấy quan hệ giữa các module vì các dependecy đều được inject vào constructor.

Khuyết điểm:

* Khái niệm DI khá khó hiểu đối với người mới tìm hiểu.
* Sử dụng interface nên đôi khi sẽ khó debug, do không biết chính xác module nào được gọi.
* Các object được khởi tạo toàn bộ ngay từ đầu, có thể làm giảm performance.
* Có thể gặp lỗi ở run-time thay vì compile-time.

**Sử dụng Dependency Injection khi nào?**

* Khi cần inject các giá trị từ một cấu hình cho một hoặc nhiều module khác nhau.
* Khi cần inject một dependency cho nhiều module khác nhau.
* Khi cần một vài service được cung cấp bởi container.
* Khi cần tách biệt các dependency giữa các môi trường phát triển khác nhau. Chẳng hạn, với môi trường dev chỉ cần log việc gửi mail, trong môi trường product cần gửi mail thông qua một API thật sự.

**Các annotation thường sử dụng**

@Component là một [Annotation](https://loda.me/java-huong-dan-tu-tao-mot-annotations-loda1554297400922) (chú thích) đánh dấu trên các Class để giúp Spring biết nó là một Bean.

@Autowired đánh dấu cho Spring biết rằng sẽ tự động inject bean tương ứng vào vị trí được đánh dấu.

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](https://loda.me/java-huong-dan-java-reflection-loda1554301692892) 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

@Primary là annotation đánh dấu trên một Bean, giúp nó luôn được ưu tiên lựa chọn trong trường hợp có nhiều Bean cùng loại trong Context.

@Qualifier xác định tên của một Bean mà bạn muốn chỉ định inject.

@Service gắn cho các Bean đảm nhiệm xử lý logic

@Repository gắn cho các Bean đảm nhiệm giao tiếp với DB

@Component gắn cho các Bean khác.

@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.

**File cấu hình và Profiles trong Spring Boot:**

Spring Boot sử dụng các file cấu hình (ví dụ: application.properties hoặc application.yml) để cấu hình ứng dụng.

Profiles trong Spring Boot cho phép bạn cấu hình ứng dụng để chạy với các tập hợp cấu hình khác nhau dựa trên môi trường hoặc yêu cầu cụ thể.

Profiles làm cho việc quản lý các cấu hình dễ dàng hơn, đồng thời giúp giảm thiểu sự phụ thuộc vào cấu hình cứng nhắc và làm cho ứng dụng trở nên linh hoạt hơn khi triển khai trên các môi trường khác nhau.

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