

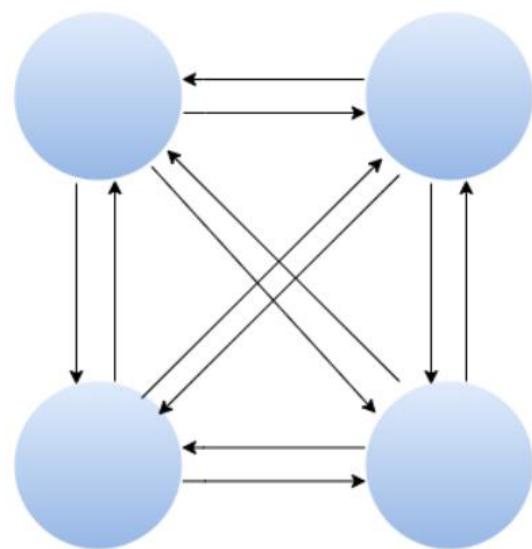


# LẬP TRÌNH JAVA SPRING BOOT

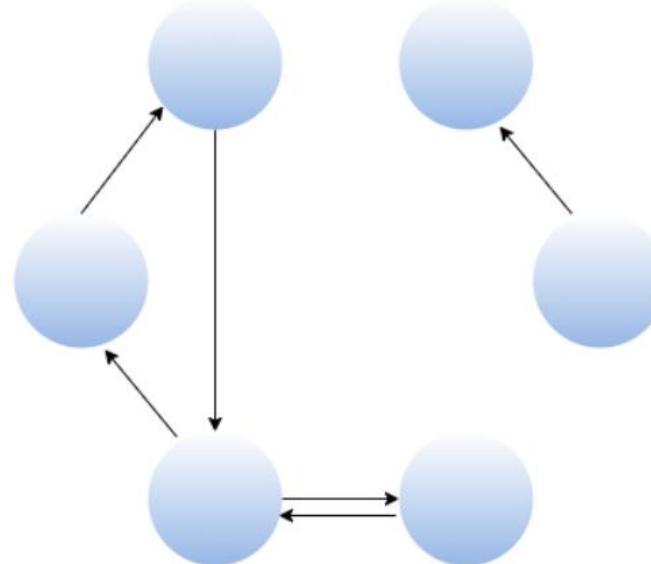
BÀI 2: MÔ HÌNH LẬP TRÌNH TRONG SPRING BOOT

- Kết thúc bài học này bạn có khả năng
  - ❖ Hiểu Tight-coupling (Liên Kết Ràng Buộc) Và Cách Loosely Coupled
  - ❖ Hiểu Dependency Injection (DI) và IoC
  - ❖ Sử dụng Các Annotations trong Spring Boot
  - ❖ Hiểu mô hình lập trình trong Spring Boot
    - Mô hình Three Tier
    - Mô hình MVC

- ❑ **Tight-coupling** 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



**Tightly Coupled**  
Many Dependencies



**Loosely Coupled**  
Some Dependencies

- ❑ Hai module **Xe hơi** và **Động cơ**, xe hơi phụ thuộc vào động cơ mới có thể chạy được. Thể hiện dạng code như sau

```
class HondaEngine { ... }

class Car {
    private HondaEngine engine;
    public Car() {
        // Khi tạo Car phải gắn engine vào
        // engine = new HondaEngine();
    }
}
```

- ❑ Sự phụ thuộc giữa Car và HondaEngine rất mạnh. Điều này dẫn tới nhiều khó khăn
  - ❖ Muốn thay đổi động cơ, cần sửa class Car
  - ❖ Không thề nào có hai Car mà sử dụng các Engine khác nhau được
  - ❖ Khó test các module hơn
- ❑ Trong chương trình không chỉ có 1, 2 module như ví dụ trên, mà có rất nhiều. Do đó, nếu các module dính quá chặt vào nhau thì sẽ rất khó bảo trì

## Coupling là gì?

- ❑ Coupling là mối quan hệ giữa hai module, hai đối tượng với nhau, có sự phụ thuộc lẫn nhau
- ❑ Và coupling có hai loại
  - ❖ Tight coupling: hai module liên kết **chặt chẽ**, khó tách rời
  - ❖ Loose coupling: liên kết yếu, **rời rạc**
- ❑ Nguyên tắc về sự phụ thuộc
  - ❖ Để code **dễ bảo trì** và **sửa đổi**, thì nguyên tắc là phải **giảm sự phụ thuộc** giữa các module
  - ❖ Nghĩa là biến mối quan hệ giữa chúng từ **tight coupling** thành **loose coupling**

- ❑ Đây là **nguyên lý** số 5 trong **SOLID** principles, tương ứng với chữ **D**.  
Được đưa ra để thiết kế các module trong chương trình, sao cho có ít sự phụ thuộc nhất có thể
- ❑ Nguyên Lý có hai ý chính
  - ❖ Các module cấp cao **không** nên phụ thuộc (trực tiếp) vào module cấp thấp. Cả hai nên phụ thuộc vào **abstraction** (của OOP)
  - ❖ Abstraction **không** nên phụ thuộc vào chi tiết, mà **ngược lại**

```
class HondaEngine { ... }
class Car {
    private HondaEngine engine;
    public Car() {
        // Khi tạo Car phải gắn engine vào
        // engine = new HondaEngine();
    }
}
```



### Bạn cần hiểu dependency là gì

- ❑ Ví dụ ở trên class Car phụ thuộc vào class HondaEngine, nên HondaEngine là một dependency (phụ thuộc) của Car. Lúc này, ta nói Car là module cấp cao, HondaEngine là module cấp thấp.

- ❑ Ví dụ ở trên code đã **vô phạm** ý 1 của DI
- ❑ Lý do là vì Car đã trực tiếp **phụ thuộc** vào Honda (do trong code class Car có sử dụng tới HondaEngine)
- ❑ Để **đúng** với DI, chúng ta sửa lại như sau. Bằng cách cho cả hai module cùng **phụ thuộc** vào **abstraction** (trong OOP thường là interface)

```
// Interface đại diện cho mọi loại động cơ
interface Engine { ... } // HondaEngine là một loại Engine

class HondaEngine implements Engine { ...
} // Trong Car thì chỉ dùng Engine (chung chung), không có cụ thể loại nào
// Loại engine cụ thể sẽ được inject vào lúc tạo (không phải gán cứng trong code)
// Do đó có thể tạo Car với các loại Engine khác nhau

class Car {
// Loại engine nào đó, lợi dụng tính đa hình OOP private Engine engine;
// Khi tạo Car thì tạo Engine object trước, rồi inject vào constructor này public
// Car(Engine engine) { this.engine = engine; }
}
```

- ❑ Abstraction chỉ lấy những thuộc tính, những hành động chung nhất, mà **không** cần quan tâm **chi tiết** bên trong chúng hoạt động thế nào
- ❑ Ví dụ về Engine, chúng ta chỉ cần biết abstraction Engine có method là run, còn những loại động cơ khác nhau thực hiện run như thế nào (chi tiết) thì không cần quan tâm

```
// Mọi loại Engine đều có thể run
```

```
interface Engine { void run();  
}
```

```
// Động cơ HondaEngine run theo kiểu khác
```

```
class HondaEngine implements Engine {public void run() {  
} // Run ổn định, bền, ít tốn xăng
```

```
// Động cơ YamahaEngine run theo kiểu khác
```

```
class YamahaEngine implements Engine {public void run() {  
} // Run nhanh, mượt nhưng tốn xăng
```

- ❑ Bên trong class Car, không quan tâm tới động cơ chạy như thế nào. Nó chỉ cần biết khi làm 1 số thao tác thì xe sẽ chạy

- ❑ Nguyên lý **không** nói cách sẽ **thực hiện** như thế nào?
- ❑ Nếu chương trình có nhiều module, nhiều object thì sẽ gặp tình trạng
  - ❖ Gắn **thiếu** module vào module khác
  - ❖ Phải quan tâm tới **thứ tự** khởi tạo module (tạo module nào trước)
  - ❖ Phụ thuộc **vòng** (A phụ thuộc B, và B cũng phụ thuộc A, suy ra không biết tạo A hay B trước)
- ❑ Do các nhược điểm trên, hiện nay người ta đưa ra khái niệm IoC - Inversion of Control - đảo ngược sự điều khiển
- ❑ Do quá trình này phức tạp và khó implement, nên đã có nhiều framework ra đời hỗ trợ IoC, điển hình như Spring cho Java hoặc Angular của JavaScript

- ❑ 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 làm nhiệm vụ **tạo, quản lý** các đối tượng trong chương trình. Phân tích các **mối** phụ thuộc
- ❑ Ví dụ:
  - ❖ Khi có nhiều Dependency (>10) chúng ta phải **tiêm** (Injection) cho các dependency này thì rất **mất thời gian**
  - ❖ Thay vì cách khởi tạo các Object như cách thông thường thì ta đảo ngược lại chiều điều khiển để cho Spring tạo Object, quản lí bộ nhớ cho các Object thay việc của chúng ta. (**ĐẢO NGƯỢC CHIỀU ĐIỀU KHIỂN**)

- ❑ Khi không dùng IoC chúng ta phải **new dependency** để khởi tạo vào class

```
Engine goodEngine = new VNEngine();
Car myCar = new Car(goodEngine);
```

- ❑ Khi áp dụng IoC trong Spring

```
@Component
class VNEngine implements Engine {
    ...
}
```

```
@Component
class Car {
    // Tìm object tương ứng với Engine và chèn (inject) vào đây
    @Autowired
    private Engine engine;
}
```

- ❑ Diễn giải: 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

- ❑ DI là một dạng **thực hiện** của IoC, bằng cách **tiêm** (inject) module vào một module khác cần nó
- ❑ DI là một **phương pháp** lập trình, là một thiết kế để bạn có được **hiệu quả** cao hơn khi code
- ❑ Khi tạo module, 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**
- ❑ Có hai loại injection
  - ❖ **Constructor-based injection:** Dùng inject các module **bắt buộc**. Các module được inject nằm trong **constructor**
  - ❖ **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

```
@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;
    }
}
```

- ❑ 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, để Spring khai báo class đó là **bean**. Ngoài **@Component**, còn có các annotation khác như **@Repository**, **@Controller**, **@Service**,... cũng được đánh dấu là **bean**.
- ❑ 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

- ❑ **Annotation** trong Spring Boot là một dạng **siêu dữ liệu** cung cấp dữ liệu về một program. Nói cách khác, annotation được sử dụng để **cung cấp thông tin** bổ sung về một program. Nó **không có ảnh hưởng** trực tiếp đến hoạt động của code. Nó **không thay đổi** hành động của chương trình đã biên dịch
- ❑ Một số Annotation trong Spring Boot
  - ❖ **@Component**: là một Annotation (chú thích) đánh dấu trên các Class để giúp Spring biết nó là một Bean
  - ❖ **@Autowired**: dùng để inject (tiêm) một bean vào class nào đó
    - Tất cả **Bean** được quản lý trong **ApplicationContext** đều chỉ được tạo ra một lần **duy nhất** và khi có class yêu cầu **@Autowired** thì nó sẽ lấy đối tượng có sẵn trong **ApplicationContext** để **inject** vào
    - Trường hợp bạn muốn mỗi lần sử dụng là một instance **mới**. Thì hãy đánh dấu **@Component** đó bằng **@Scope("prototype")**

- ❑ Trong thực tế, sẽ có trường hợp chúng 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
- ❑ Ví dụ

```
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 Sweater implements Outfit {
    @Override
    public void wear() {
        System.out.println("Mặc sweater");
    }
}
```

```
@Component
public class Jean implements Outfit {
    @Override
    public void wear() {
        System.out.println("Đang mặc quần jean");
    }
}
```

- ❑ Cách giải quyết **thứ nhất** là sử dụng Annotation **@Primary**
- ❑ **@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

```
@Component  
@Primary  
public class Sweater implements Outfit {  
    @Override  
    public void wear() {  
        System.out.println("Mặc sweater");  
    }  
}
```

- ❑ Class Girl yêu cầu inject một Outfit vào cho mình

```
@Component  
public class Girl {  
  
    @Autowired  
    Outfit outfit;  
  
    // GET  
    // SET  
}
```

## ☐ 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: com.likelion.threetier.service.Girl@41477a6d
Girl Outfit: com.likelion.threetier.service.impl.Sweater@2bc12da
Mặc sweater
```

- ❑ Cách giải quyết **thứ hai** là sử dụng Annotation **@Qualifier**
- ❑ **@Qualifier** xác định tên của một Bean mà bạn muốn chỉ định inject

```
@Component("sweater")
public class Sweater implements Outfit {
    @Override
    public void wear() {
        System.out.println("Mặc sweater");
    }
}
@Component("jean")
public class Jean implements Outfit {
    @Override
    public void wear() {
        System.out.println("Đang mặc quần jean");
    }
}
@Component
public class Girl {
    @Autowired
    @Qualifier("jean")
    Outfit outfit;
    // GET
    // SET
}
```

Chỉ định tên Bean bằng  
    @Qualifier("jean")

- ❑ **@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

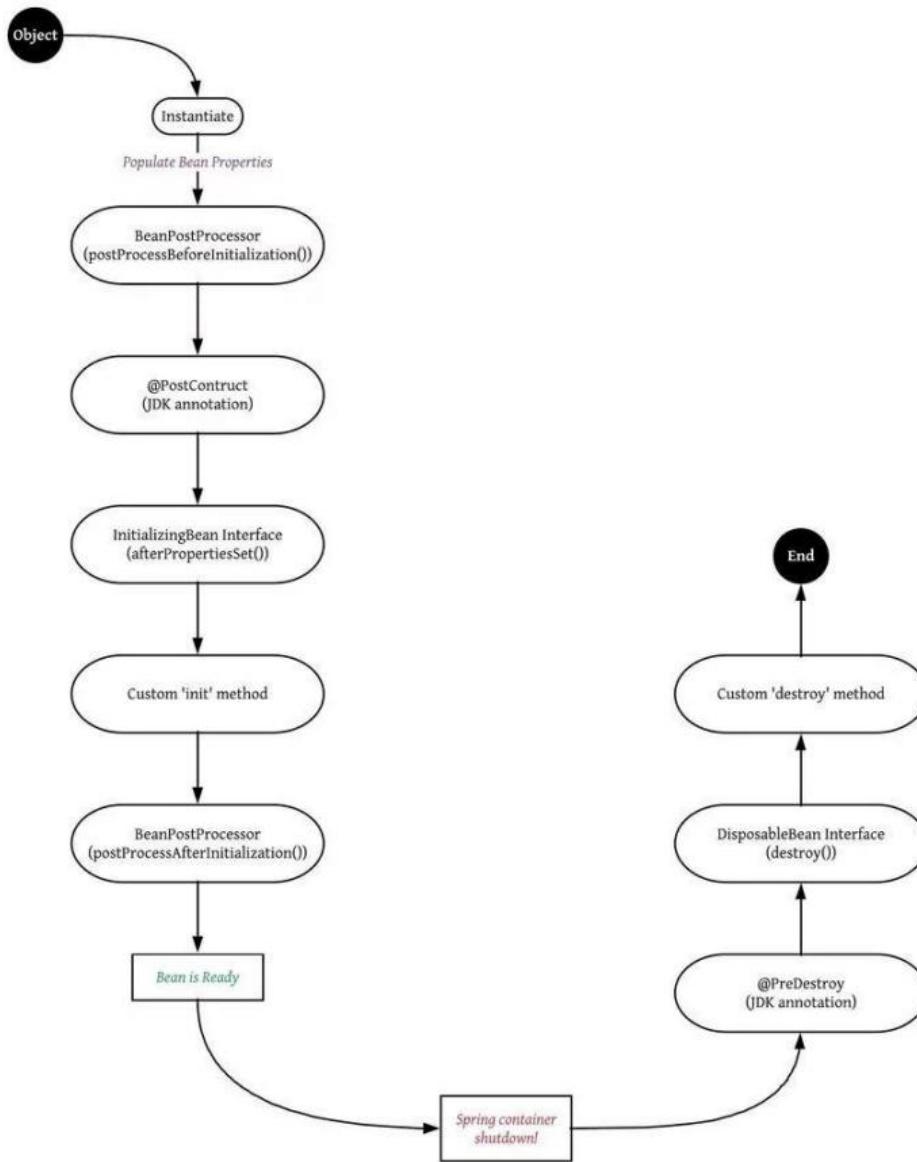
```
@Component
public class Girl {

    @PostConstruct
    public void postConstruct(){
        System.out.println("\t>> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này");
    }
}
```

- ❑ **@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 postConstruct(){
        System.out.println("\t>> Đối tượng Girl trước khi bị destroy thì chạy hàm này");
    }
}
```

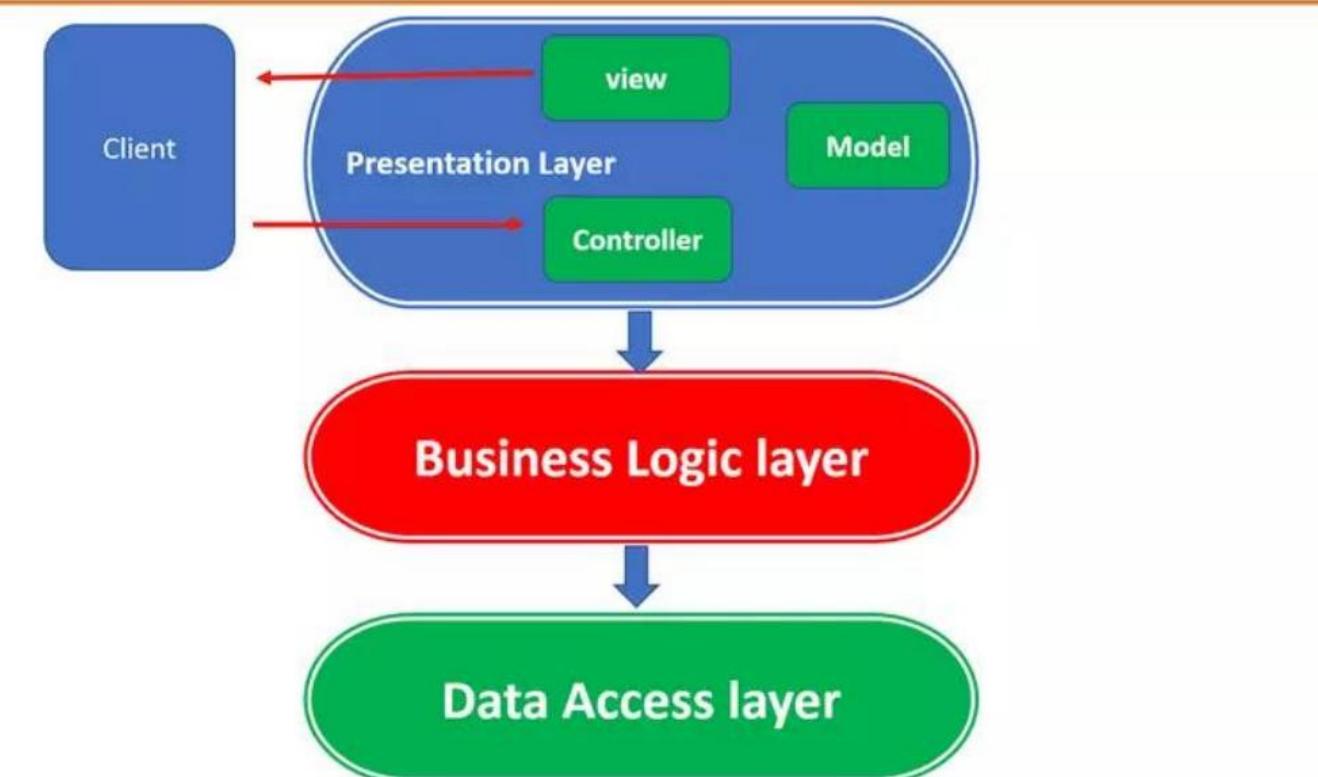


- ❑ Nhìn có vẻ loằng ngoằng, Bạn có lẽ sẽ chỉ cần hiểu căn bản như sau
  - ❖ Khi IoC Container (ApplicationContext) tìm thấy một Bean cần quản lý, nó sẽ khởi tạo bằng Constructor
  - ❖ inject dependencies vào Bean bằng Setter, và thực hiện các quá trình cài đặt khác vào Bean như setBeanName, setBeanClassLoader, v.v..
  - ❖ Hàm đánh dấu @PostConstruct được gọi
  - ❖ Tiền xử lý sau khi @PostConstruct được gọi
  - ❖ Bean sẵn sàng để hoạt động
  - ❖ Nếu IoC Container không quản lý bean nữa hoặc bị shutdown nó sẽ gọi hàm @PreDestroy trong Bean
  - ❖ Xóa Bean

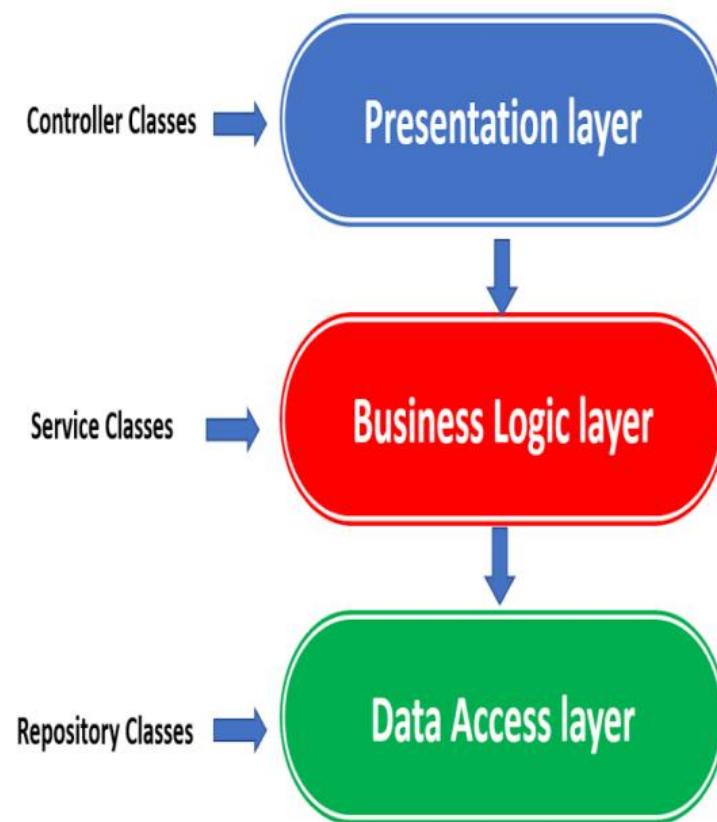
## ❑ Hai mô hình quen thuộc

- ❖ Mô hình 3 lớp (three tier)
- ❖ Mô hình MVC

## Three-Tier architecture vs MVC pattern



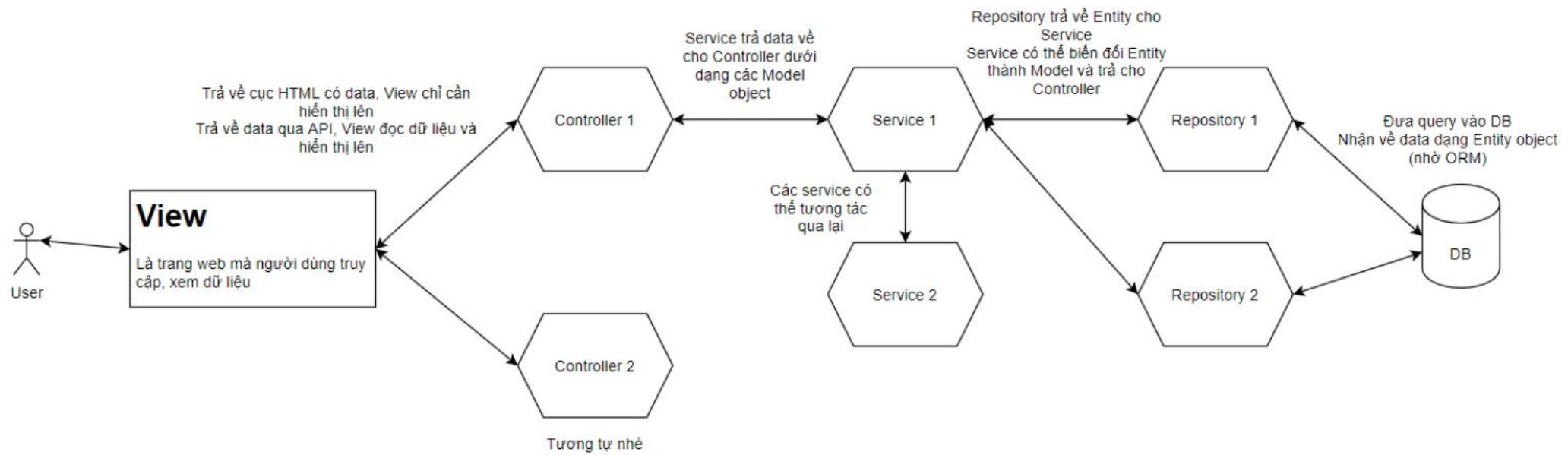
- ❑ Spring Boot tuân theo một kiến trúc **phân lớp**, trong đó mỗi lớp **giao tiếp** với lớp ngay bên dưới hoặc bên trên (cấu trúc phân cấp) nó
- ❑ Có ba lớp trong Spring Boot như sau
  - ❖ **Presentation:** Tầng này tương tác với người dùng, bằng View, Controller (trong MVC) hoặc API (nếu có)
  - ❖ **Service:** Chứa toàn bộ logic của chương trình, các đa số code nằm ở đây
  - ❖ **Repository:** Tương tác với database, trả về kết quả cho tầng business logic



- ❑ Được xây dựng dựa trên tư tưởng **độc lập**, ám chỉ việc các layer phục vụ các mục đích **nhất định**, khi muốn thực hiện một công việc **ngoài phạm vi** thì sẽ đưa công việc xuống các layer **thấp hơn**
- ❑ Để phục vụ cho kiến trúc **three tier** Spring Boot tạo ra 3 Annotation là **@Controller** và **@Service** và **@Repository** để chúng ta có thể đánh dấu các tầng với nhau
  - ❖ **@Controller:** Là tầng giao tiếp với bên ngoài và handler các request từ bên ngoài tới hệ thống
  - ❖ **@Service:** Đánh dấu một Class là tầng Service, phục vụ các logic nghiệp vụ. Chứa các business logic code
  - ❖ **@Repository:** Đánh dấu một Class Là tầng Repository, phục vụ truy xuất dữ liệu. Đại diện cho tầng data access

- ❑ Do Spring Boot chỉ là wrapper cho Spring, chúng ta vẫn sử dụng **ngầm** các module Spring khác bên dưới
- ❑ Khi dùng Spring MVC phải **tuân theo** mô hình MVC
- ❑ Mô hình **MVC** chia làm 3 phần
  - ❖ **Model**: các cấu trúc dữ liệu của toàn chương trình, có thể đại diện cho trạng thái của ứng dụng
  - ❖ **View**: lớp giao diện, dùng để hiển thị dữ liệu ra cho user xem và tương tác
  - ❖ **Controller**: kết nối giữa Model và View, điều khiển dòng dữ liệu
- ❑ Dữ liệu từ Model qua Controller sau đó được gửi cho View hiển thị ra. Và ngược lại, khi có yêu cầu mới từ View, thì sẽ qua Controller thực hiện thay đổi dữ liệu của Model
- ❑ **Lưu ý**: MVC chỉ mô tả **luồng đi** của dữ liệu, nó **không** nói rõ như code đặt ở đâu (ở Model, View hay Controller), lưu trữ Model vào database như thế nào,... Do đó, đối với ứng dụng **hoàn chỉnh** như Spring Boot thì cần **kết hợp** cả mô hình **MVC** và **3-tier** lại với nhau

## ❑ Sơ đồ luồng đi





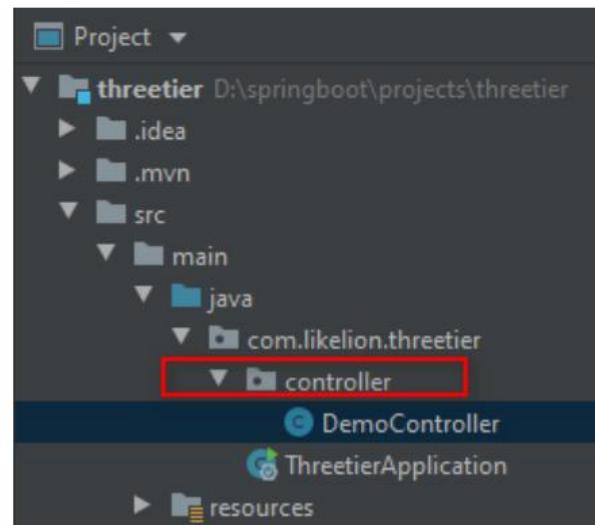
- ☐ Truy cập <https://start.spring.io/> để tạo project Spring Boot

The screenshot shows the Spring Initializr interface with the following configurations:

- Project:** Maven Project (selected)
- Language:** Java (selected)
- Spring Boot:** 2.7.1 (selected)
- Dependencies:** Spring Web (WEB) selected
- Project Metadata:**
  - Group: com.likelion
  - Artifact: threetier
  - Name: threetier
  - Description: (empty)
  - Package name: com.likelion.threetier
  - Packaging: Jar (selected)
  - Java: 8 (selected)
- Bottom Buttons:** GENERATE (highlighted with a red box), EXPLORE, SHARE...

A blue callout box with the text "Lưu project về máy" is positioned over the GENERATE button.

- ❑ Mở project bằng IntelliJ và tạo package controller theo cấu trúc như hình



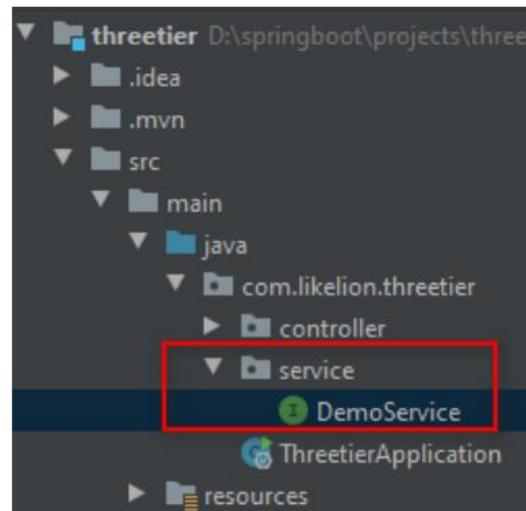
- ❑ Tiếp theo tạo file Controller như hình

Nếu người dùng request tới địa chỉ "/"

```
package com.likelion.threetier.controller;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
@RestController  
public class DemoController {  
    @GetMapping(value = "/")  
    public String getDemo() {  
        return "";  
    }  
}
```

## ☐ Tạo package service và tạo class

DemoService như hình



## ☐ Class DemoService

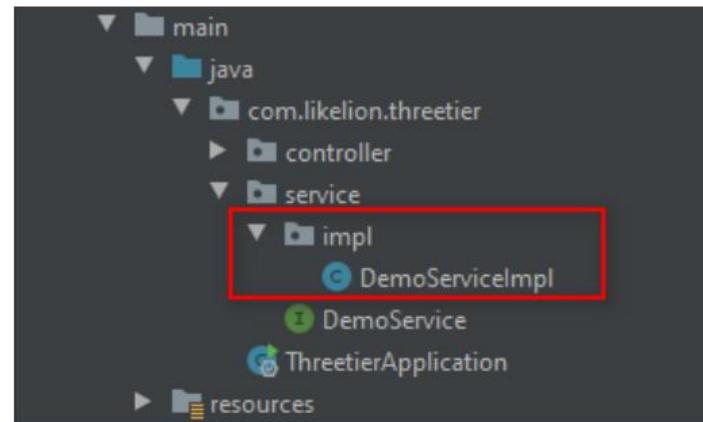
```
Project D:\springboot\projects\threetier DemoService.java
threetier D:\springboot\projects\threetier
  .idea
  .mvn
  src
    main
      java
        com.likelion.threetier
          controller
          service
            DemoService
  ThreetierApplication
  resources
```

The screenshot shows the code editor for 'DemoService.java'. The interface definition is as follows:

```
package com.likelion.threetier.service;

public interface DemoService {
```

- ❑ Tạo package impl bên trong package service và tạo class DemoServiceImpl như hình

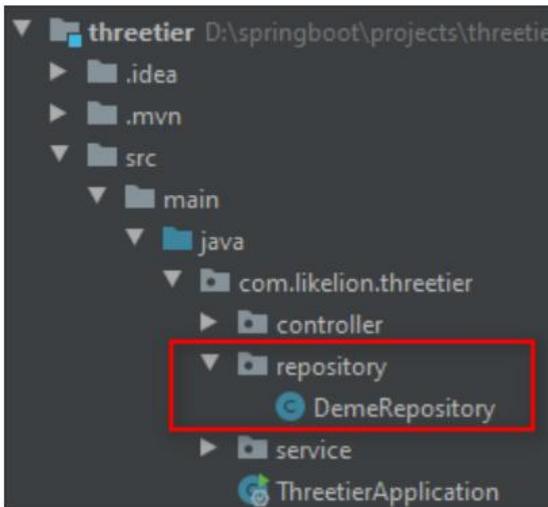


- ❑ Class DemoServiceImpl

```
Project D:\springboot\projects\threetier
src
  +-- main
    +-- java
      +-- com.likelion.threetier
        +-- controller
        +-- service
          +-- impl
            +-- DemoServiceImpl.java

DemoService.java x DemoServiceImpl.java x
1 package com.likelion.threetier.service.impl;
2
3 import com.likelion.threetier.service.DemoService;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class DemoServiceImpl implements DemoService {
8
9 }
10
```

- ❑ Tạo package repository và tạo class DemoRepository như hình



- ❑ Class DemeRepository

The image shows the IntelliJ IDEA code editor with the file 'DemeRepository.java' open. The code is as follows:

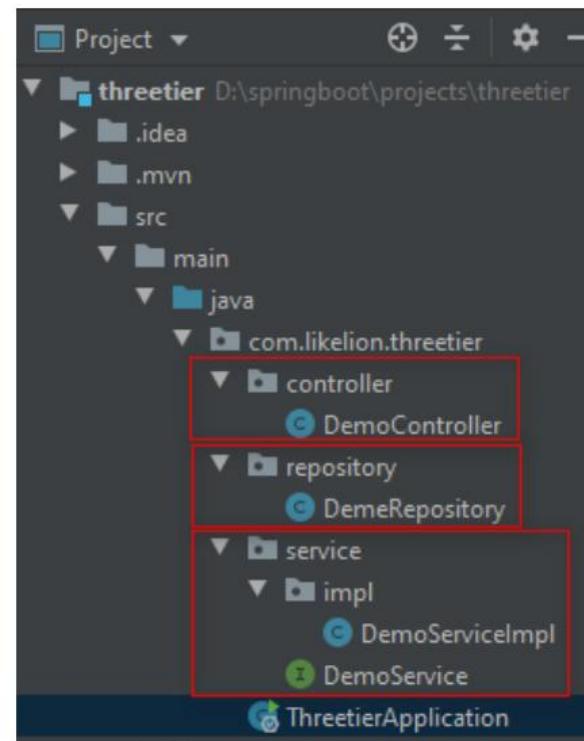
```
package com.likelion.threetier.repository;  
import org.springframework.stereotype.Repository;  
@Repository  
public class DemeRepository {  
}
```

The code editor's left pane shows the project structure, which matches the file tree shown above. The 'DemeRepository' class is annotated with '@Repository' and is highlighted with a red box.

- ❑ Các annotations **@RestController**, **@Service**, **@Repository** dùng để chú thích các tầng (layer) của mô hình

 **Tại sao tầng Service class DemoServiceImpl phải implement interface DemoService ?**

- ❑ Vì tất cả các xử lý nghiệp vụ (Business logic) sẽ được trừu tượng hoá



## ❑ Lưu ý

- ❖ **@Controller** là sẽ trả về một template
- ❖ **@RestController** trả về dữ liệu dưới dạng JSON

- ❑ Tight-coupling là một khái niệm trong Java ám chỉ việc mối quan hệ giữa các Class quá chặt chẽ
- ❑ Loosely-coupled là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các Class với nhau
- ❑ Nguyên Lý Dependency Inversion đưa ra để thiết kế các module trong chương trình sao cho có ít sự phụ thuộc nhất có thể
- ❑ 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**
- ❑ DI là một dạng **thực hiện** của IoC, bằng cách **tiêm** (inject) module vào một module khác cần nó
- ❑ Các annotations căn bản: **@Component, @Autowired, @Primary, @Qualifier, @PostConstruct, @PreDestroy, @Controller, @RestController, @Service vs @Repository**
- ❑ Luồng đi trong Spring Boot. Hai mô hình quen thuộc
  - ❖ Mô hình 3 lớp (three tier)
  - ❖ Mô hình MVC

# Cảm Ơn Bạn Đã Chăm Chỉ!

