Trường ĐH Công Nghệ Sàigòn

**KHOA CÔNG NGHỆ THÔNG TIN**

ĐỒ ÁN CHUYÊN NGÀNH

*Tên đề tài:*

TÌM HIỂU VÀ XÂY DỰNG WEBSITE THỜI TRANG BẰNG SPRING MVC

TPHCM – Năm 2019

Trường ĐH Công Nghệ Sàigòn

**KHOA CÔNG NGHỆ THÔNG TIN**

ĐỒ ÁN CHUYÊN NGÀNH

*Tên đề tài:*

Tìm hiểu và xây dựng website thời trang bằng SpringMVC

Người hướng dẫn: Ks. Lê Triệu Ngọc Đức

Sinh viên thực hiện: Nguyễn Hoàng Hải

TPHCM – Năm 2019

Mục lục

Mục lục 1

Danh sách các hình vẽ và bảng 2

Chương 1. Giới thiệu 3

1.1 Mục đích 3

1.2 Phạm vi 3

1.3 Yêu cầu 3

Chương 2. Công cụ lập trình 5

2.1 Eclipse 5

2.2 Maven 5

Chương 3. Cơ sở lý thuyết 11

3.1 Java Reflection 11

3.2 Mô hình MVC 15

3.3 Spring Framework 20

**TÀI LIỆU THAM KHẢO**

Danh sách các hình vẽ và bảng

[Hình 2‑1 Install Maven on Eclipse IDE 6](#_Toc25005113)

[Hình 2‑2 icon m2eclipse 7](#_Toc25005114)

[Hình 2‑3 Mave Project 7](#_Toc25005115)

[Hình 2‑4 Nguyên tắc hoạt động Maven 9](#_Toc25005116)

[Hình 3‑1 Kiến trúc tổng quát Java Reflection API 12](#_Toc25005117)

[Hình 3‑2 Mô hình MVC 16](#_Toc25005118)

[Hình 3‑3 Quy trình hoạt động mô hình MVC trong dự án website 17](#_Toc25005119)

[Hình 3‑4 Chi tiết về mô hình MVC 18](#_Toc25005120)

[Hình 3‑5 Luồng của một chương trình bình thường 23](#_Toc25005121)

[Hình 3‑6 Servlet Container 24](#_Toc25005122)

[Hình 3‑7 UnitTest 25](#_Toc25005123)

[Hình 3‑8 Luồng của một chương trình IoC 25](#_Toc25005124)

[Hình 3‑9 Dependency Injection 26](#_Toc25005125)

[Hình 3‑10 Kiến trúc Spring Framework 27](#_Toc25005126)

[Hình 3‑11 Spring IoC Container 33](#_Toc25005127)

[Hình 3‑12 Luồng xử lý DispatcherServlet 48](#_Toc25005128)

[Hình 3‑13 Luồng xử lý một request trong Spring MVC 49](#_Toc25005129)

# Giới thiệu

## Mục đích

Giới thiệu mục đích của việc thực hiện đề tài phân tích thiết kế: *để hỗ trợ / tin học hoá công việc/ phân tích và thiết kế cho ứng dụng để phục vụ mục đích gì*

## Phạm vi

Nêu những giới hạn phạm vi ứng dụng của đề tài

## Yêu cầu

Để có kết quả tốt nhất khi tham khảo báo cáo này, cá nhân nên có kiến thức cơ bản về ngôn ngữ lập trình java, sử dụng và thao tác được với framework ORM hibernate, JPA, hiểu về vòng đời và biết cách tạo ra servlet và jsp.

Để có kết quả tốt nhất khi tham khảo “chương trình demo đồ án” này, ngoài những yêu cầu trên thì cá nhân cũng cần hiểu về các thư viện khi thao tác với jsp như (jstl, tags-form, apache tiles framework), spring data jpa và lib lombok. Những yêu cầu khi tham khảo chương trình demo đồ án không nằm trong phạm vi đồ án này, nó chỉ giúp quá trình xây dựng và phát triển chương trình demo được gọn và nhanh chóng hơn.

# Công cụ lập trình

## Eclipse

## Maven

Sau đây là giới thiệu sơ lược về Maven cũng như cách cài đặt và thêm thư viện vào Maven Project.

### Tổng quan

Maven là công cụ quản lý và thiết lập tự động 1 dự án phần mềm. Chủ yếu dùng cho các lập trình viên java, nhưng nó cũng có thể được dùng để xây dựng và quản lý các dự án dùng C#, Ruby, Scala hay ngôn ngữ khác. Maven phục vụ mục đích tương tự như Apache Ant, nhưng nó dựa trên khái niệm khác và cách hoạt động khác. Maven được quản lý bởi Apache Software Foundation, nó là một phần của dự án Jakarta Project.

Maven dùng khái niệm Project Object Model (POM) để mô tả việc build project, các thành phần phụ thuộc và các module. Nó định nghĩa trước các target cho việc khai báo task, trình biên dịch, đóng gói và thứ tự hoạt động để mọi việc diễn ra tốt nhất.

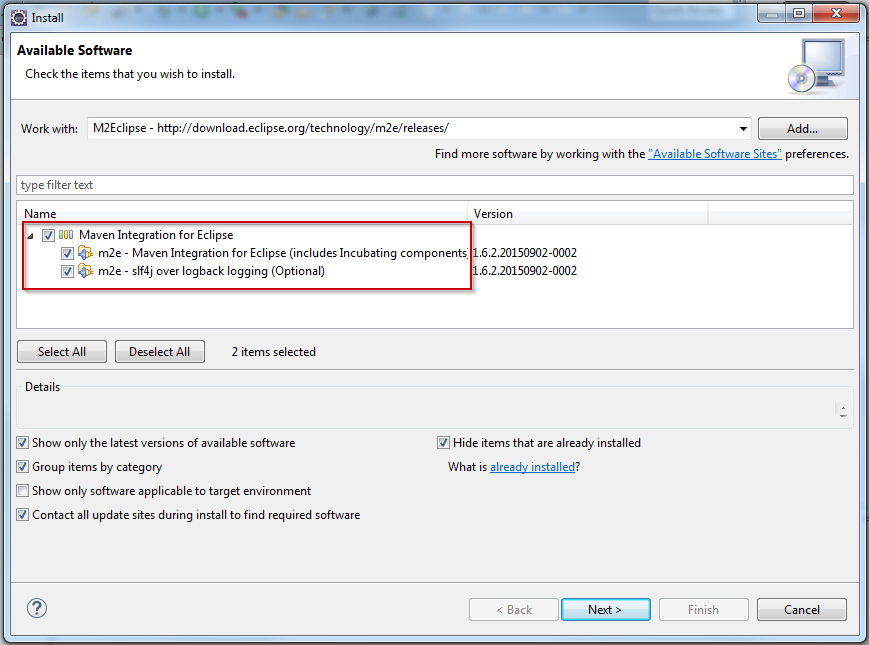
Maven tải các thư viện, plug-in từ 1 hoặc nhiều repositories. Maven cung cấp việc hỗ trợ tải file từ repository trung tâm của Maven và các repositories khác. Cũng có thể upload các sản phẩm lên repository sau khi đã build xong.

### Cài đặt Maven

Hầu hết các bản tải xuống Eclipse đều bao gồm công cụ Maven, nếu nó bị thiếu trong trình cài đặt, chúng ta có thể cài đặt nó như một Plugin nhúng vào Eclipse.

Sau đây là từng bước hướng dẫn cài Maven trên Eclipse IDE:

* Mở Eclipse IDE và chọn Help -> Install New Software…
* Ở mục Work with điền vào biểu mẫu với thông tin như bên dưới.
  + <http://download.eclipse.org/technology/m2e/releases>
* Sau khi chờ kết thúc, chọn tất cả các Plugin hiện ra và nhấn Next.

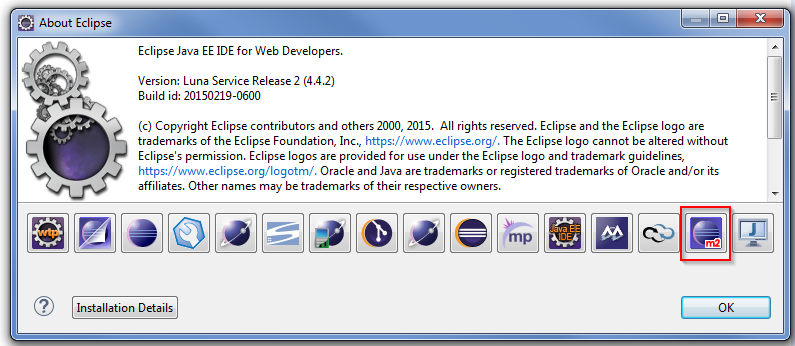


Hình 2‑1 Install Maven on Eclipse IDE

* Chấp nhận các điều khoản của thỏa thuận cấp phép và nhấn vào Finish.
* Sau khi kết thúc cài đặt, chúng ta sẽ được yêu cầu khởi động lại Eclipse. Chọn Yes để thực hiện khởi động lại.

Để kiểm tra Maven đã hoàn thành cài đặt hay chưa ta làm như sau:

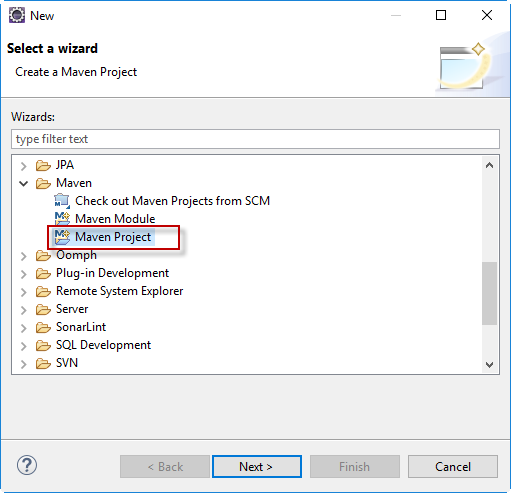
* Mở Eclipse IDE và chọn Help -> About Eclipse
* Kiểm tra sự hiện diện của icon M2Eclipse như bên dưới



Hình 2‑2 icon m2eclipse

### Tạo Project Maven

* Mở Eclipse IDE -> File -> New -> Maven Project



Hình 2‑3 Mave Project

* Trong Dialog hiện ra, lựa chọn “Create a simple project…” nếu chúng ta muốn tạo một dự án đơn giản. Lựa chọn “Use default Workspace location” cho phép chúng ta tạo ra dự án theo một template mẫu. Sau đó nhấn Next.
* Nhập các giá trị
  + Group Id: tên\_tổ\_chức.công\_ty.cá\_nhân của dự án. Ví dụ com.nguyenhai.abc
  + Artifact Id: tên dự án.
  + Version: phiên bản của dự án.
  + Package: chọn jar nếu dự án là Java Application, war nếu là Web Application.
  + Name: tên của dự án trong Eclipse
* Nhấn Finish để kết thúc quá trình tạo Project Maven.

### Cấu hình Maven

File pom.xml là nơi khai báo tất cả những gì liên quan đến dự án được cấu hình qua maven, như khai báo các dependency, version của dự án, tên dự án, repository, … Mở file pom.xml ra, chúng ta thấy nội dung như sau:

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0*

*http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>stu.hoanghai.abc</groupId>

<artifactId>SpringCore</artifactId>

<version>0.0.1-SNAPSHOT</version>

</project>

Tất nhiên là các mục nội dung sẽ khác, dựa vào thông tin ta đã khai báo trong quá trình tạo Project Maven.

Để thêm một thư viện vào Project Maven chúng ta có thể truy cập vào đường dẫn sau https://mvnrepository.com/, tìm kiếm và copy Dependency của thư viện đó và thêm vào file pom.xml như sau:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.2.0.RELEASE</version>

</dependency>

</dependencies>

Tag dependencies sẽ chứa tất cả những khai báo dependency của các thư viện mà chúng ta sử dụng, dependencies là 1 tag con trong tag project.

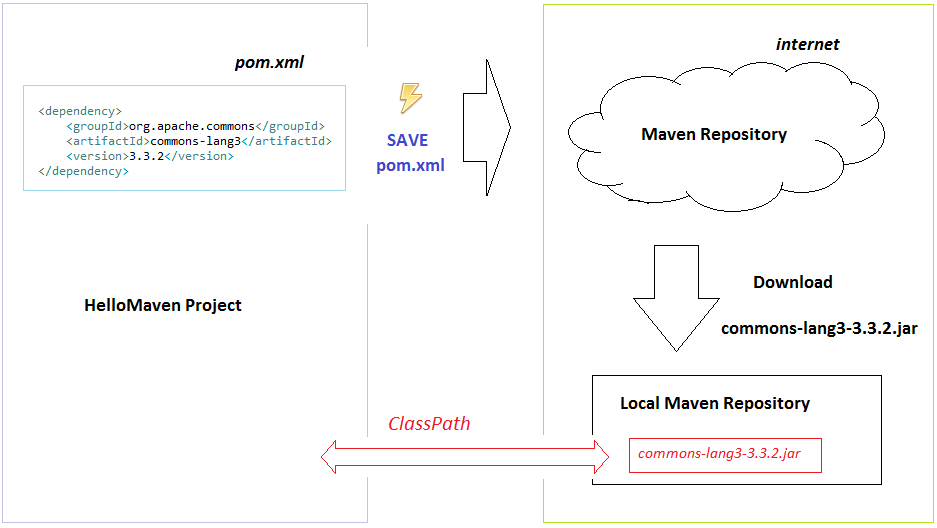
Lưu lại file pom.xml và chờ đợi project build xong. Thư viện sẽ được thêm tự động vào Project ở mục Maven Dependencies.

### Nguyên tắc hoạt động

Khi khai báo pom.xml rằng Project của ta phụ thuộc vào thư viện nào đó.

Ngay sau khi lưu, maven sẽ tiến hành kiểm tra xem thư viện này đã tồn tại ở kho chứa địa phương chưa (Local Repository), nếu chưa có nó sẽ tiến hành download về từ kho chứa (Reposistory) thư viện từ trên Internet xuống.

Cuối cùng, Maven sẽ tự động khai báo ClassPath cho Project tới vị trí file jar được tải về.



Hình 2‑4 Nguyên tắc hoạt động Maven

# Cơ sở lý thuyết

## Java Reflection

Java Reflection là nền tảng cho cách thức làm việc của tất cả các dependency injection framework trong java như Dagger, Google Guice, PicoContainer và Sping, chính vì vậy việc tìm hiểu về Java reflection sẽ giúp chúng ta dễ dàng tiếp cận Spring hơn.

Reflection là kĩ thuật rất cần thiết để lấy các thông tin của một kiểu dữ liệu. Dựa vào đó ta có thể kích hoạt (như các phương thức) hoặc tạo thể hiện của kiểu dữ liệu đó. Một ứng dụng quan trọng của reflection mà ta có thể biết là Java Bean. Nhờ đó, các IDE (như NetBeans) có thể lấy được các thông tin và thiết lập giá trị cho các đối tượng.

Lưu ý: Các framework dependency injection đã hỗ trợ gần như triệt để kỹ thuật này. Vì vậy chúng ta sẽ không đi sâu vào tìm hiểu java reflection mà chỉ dừng lại ở mức hiểu cách thức mà nó hoạt động.

### Kiến trúc của Java Reflection API

Các lớp được dùng trong reflection nằm trong hai package là java.lang và java.lang.reflect. Package java.lang.reflect bao gồm ba lớp chính mà chúng ta cần biết là Constructor, Field và Method:

* Class<T>: lớp này đại diện cho các class, interface và chứa các phương thức dùng để lấy các đối tượng kiểu Constructor, Field, Method,…
* AccessibleObject: các kiểm tra về phạm vi truy xuất (public, private, protected) của field, method, constructor sẽ được bỏ qua. Nhờ đó ta có thể dùng reflection để thay đổi, thực thi các thành phần này mà không cần quan tâm đến phạm vi truy xuất của nó.
* Constructor: chứa các thông tin về một constructor của lớp.
* Field: chứa các thông tin về một field của class, interface.
* Method: chứa các thông tin về một phương thức của class, interface.

Hình 3‑1 Kiến trúc tổng quát Java Reflection API

### Tạo đối tượng Class<>

Đối tượng kiểu này được tạo ra bằng cách sử dụng phương thức static Class.forName().

**try** {

Class c = Class.*forName*("entity.User");

} **catch** (ClassNotFoundException e) {

System.***err***.println(e);

}

Trong trường hợp không tìm thấy lớp tương ứng phương thức trên sẽ ném ra ngoại lệ ClassNotFoundException.

Hoặc chúng ta có thể dùng cách khác với từ class (Class literal) cách này đảm bảo rằng lớp được sử dụng luôn luôn tồn tại và không có ngoại lệ nào bị ném ra cả.

Class<User> user = User.**class**;

Class<Foo> foo = Foo.**class**;

Dưới đây là một ví dụ đơn giản để in ra thông tin của một lớp trong môi trường run time chỉ cần chúng ta biết được tên đầy đủ (bao gồm cả phần Package) của lớp đó, giả sử chúng ta có lớp User như sau:

**package** entity;

**public** **class** User {

**private** String username;

**private** String password;

**public** User() {

**super**();

}

**public** **void** printInfo() {

System.***out***.println(username + " " + password);

}

// get and set

}

Hàm main như sau:

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

Class<?> c = Class.*forName*("entity.User");

System.***out***.print(c.getName() + " ");

System.***out***.print(c.getSimpleName() + " ");

**for**(Method method : c.getDeclaredMethods()) {

System.***out***.print(method.getName() + " ");

}

}

Kết quả

entity.User User setUsername setPassword getPassword getUsername printInfo

### Lấy và gán giá trị cho Field

Để truy xuất đến một thành viên của lớp nói chung (field, method…), ta chỉ cần lấy được đối tượng đại diện tương ứng thông qua các phương thức của Class. Sau đó sử dụng các phương thức của đối tượng đại diện để thao tác.

Trong ví dụ này, ta sẽ sử dụng những phương thức sau:

* Class.getField(String name): trả về một đối tượng Field.
* Class.getDeclaredField(String name): trả về một đối tượng Field là private.
* Field.set(Object obj, Object value): gán value cho field tương ứng của đối tượng obj.

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

User user = **new** User();

user.setUsername("HoangHai");

user.setPassword("123-STU");

System.***out***.println(user.getUsername() +

" password = " + user.getPassword());

Class<?> c = Class.*forName*("entity.User");

Field field = c.getDeclaredField("password");

field.setAccessible(**true**);

field.set(user, "STU-123");

System.***out***.println(user.getUsername() +

" password = " + user.getPassword());

}

Kết quả

HoangHai password = 123-STU

HoangHai password = STU-123

### Thực thi một phương thức

Để kích hoạt một phương thức cụ thể, ta cần sử dụng hai phương thức instance sau:

* Class.getMethod(String name, Class[] parameterTypes): trả về đối tượng Method đại diện cho một phương thức của lớp. Phương thức này được xác định qua tên và các kiểu tham số.
* Method.invoke(Object obj, Object[] args): thực thi phương thức tương ứng của đối tượng obj với các tham số args.

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

Class<?> foo = Class.*forName*("entity.Foo");

Method method = foo.getMethod("add", **int**.**class**, **int**.**class**);

Object result = method.invoke(**new** Foo(), 3, 8);

System.***out***.println("3 + 8 = " + result);

}

Kết quả

3 + 8 = 11

### Tạo sự thể hiện của lớp

Có hai phương thức để tạo một thể hiện của lớp:

* Class.newInstance(): tạo một đối tượng với constructor không có tham số.
* Constructor.newInstance(Object[] initargs): tạo đối tượng với constructor có tham số.

Ví dụ

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

Class<?> foo = Class.*forName*("entity.User");

User user = (User)foo.newInstance();

user.setUsername("HoangHai");

user.setPassword("STU-123");

user.printInfo();

}

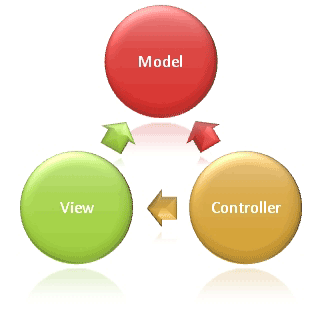
Kết quả

HoangHai STU-123

## Mô hình MVC

### Khái niệm

Mô hình MVC là một chuẩn mô hình và đóng vai trò quan trọng trong quá trình xây dựng – phát triển – vận hành và bảo trì một hệ thống hay một ứng dụng – phần mềm. Nó tạo ra một mô hình 3 lớp Model – View – Controller tách biệt và tương tác nhau, giúp các chuyên gia có thể dễ dàng dựa vào mô hình để trao đổi và xử lý những nghiệp vụ một cách nhanh chóng. Đây là một mô hình đã xuất hiện từ những năm 70 của thế kỷ 20 tại phòng thí nghiệm Xerox PARC ở Palo Alto, nó không phụ thuộc vào môi trường, nền tảng xây dựng hay ngôn ngữ phát triển. Chúng ta có thể áp dụng mô hình MVC vào các dự án trong môi trường Windows, Linux… và sử dụng bất kỳ ngôn ngữ nào như PHP, ASP, JSP…



Hình 3‑2 Mô hình MVC

Mô hình MVC được chia làm 3 lớp xử lý gồm Model – View – Controller:

* Model: là nơi chứa những nghiệp vụ tương tác với dữ liệu hoặc hệ quản trị cơ sở dữ liệu (mysql, mssql…), nó sẽ bao gồm các class/function xử lý nhiều nghiệp vụ như kết nối database, truy vấn dữ liệu, thêm – xóa – sửa dữ liệu…
* View: là nới chứa những giao diện như một nút bấm, khung nhập, menu, hình ảnh… nó đảm nhiệm nhiệm vụ hiển thị dữ liệu và giúp người dùng tương tác với hệ thống.
* Controller: là nới tiếp nhận những yêu cầu xử lý được gửi từ người dùng, nó sẽ gồm những class/ function xử lý nhiều nghiệp vụ logic giúp lấy đúng dữ liệu thông tin cần thiết nhờ các nghiệp vụ lớp Model cung cấp và hiển thị dữ liệu đó ra cho người dùng nhờ lớp View.

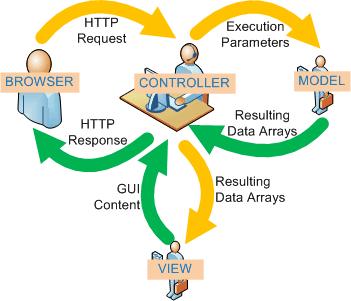
### Sự tương tác giữa các lớp trong mô hình MVC

Controller – View sẽ lấy những hình ảnh, nút bấm… hoặc hiển thị dữ liệu được trả ra từ Controller để người dùng có thể quan sát và thao tác. Trong sự tương tác này cũng có thể không có dữ liệu được lấy từ Model và khi đó nó chỉ chịu trách nhiệm hiển thị đơn thuần như hình ảnh, nút bấm…

Controller – Model là luồng xử lý khi controller tiếp nhận yêu cầu và các tham số đầu vào từ người dùng, controller sẽ sử dụng các lớp/hàm trong Model cần thiết để lấy ra những dữ liệu chính xác.

View – Model có nhiều cách hiểu về sự tương tác giữa 2 thành phần này. Hoặc, chúng có thể tương tác với nhau mà không thông qua controller khi chỉ đơn thuần là hiển thị dữ liệu tĩnh và không xử lý nghiệp vụ logics nào, hoặc là không hề có sự tương tác nào giữa hai 2 thành phần này, cho dù có hay không có xử lý nghiệp vụ logics thì vẫn thông qua controller.

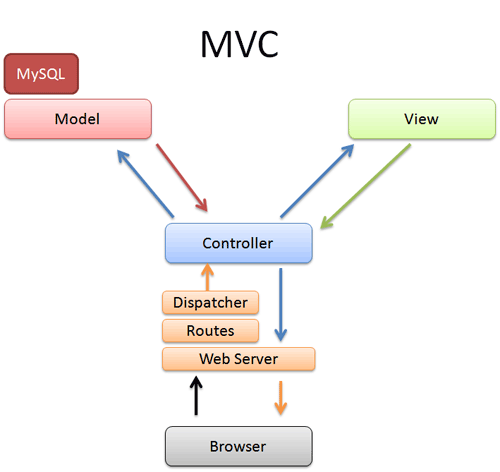
### Quy trình hoạt động mô hình MVC trong dự án website



Hình 3‑3 Quy trình hoạt động mô hình MVC trong dự án website

Mô hình trên là thể hiện mô hình MVC trong các dự án website và sẽ hoạt động theo quy trình sau:

* Người dùng sử dụng một BROWSER trình duyệt web để có thể gửi những yêu cầu (HTTP Request) có thể kèm theo những dữ liệu nhập tới những CONTROLLER xử lý tương ứng. Việc xác định Controller xử lý sẽ dựa vào một bộ Routing điều hướng.
* Khi CONTROLLER nhận được yêu cầu gửi tới, nó sẽ chịu trách nhiệm kiểm tra yêu cầu đó có cần dữ liệu từ MODEL hay không? Nếu có, nó sẽ sử dụng các phương thức cần thiết trong MODEL và nó sẽ trả ra kết quả (Resulting Arrays), khi đó CONTROLLER sẽ xử lý giá trị đó và trả ra VIEW để hiển thị. CONTROLLER sẽ xác định các VIEW tương ứng để hiển thị đúng với yêu cầu.
* Khi nhận được dữ liệu từ CONTROLLER, VIEW sẽ chịu trách nhiệm xây dựng các thành phẩn hiển thị như hình ảnh, thông tin dữ liệu… và trả về GUI Content để CONTROLLER đưa ra kết quả lên màn hình BROWSER.
* BROWSER sẽ nhận giá trị trả về (HTTP Response) và sẽ hiển thị với người dùng. Kết thúc một quy trình hoạt động.



Hình 3‑4 Chi tiết về mô hình MVC

### Ưu điểm và nhược điểm của mô hình MVC

Ưu điểm:

* Các dự án có thể áp dụng ngay mô hình MVC mà không phụ thuộc môi trường, nền tảng xây dựng hay ngôn ngữ lập trình phát triển.
* Quy hoạch các class/ function vào các thành phần riêng biệt Controller – Model – View, khi đó sẽ dễ dàng xây dựng – phát triển – quản lý – vận hành và bảo trì một dự án, tạo sự rõ ràng, trong sáng trong quá trình phát triển dự án, kiểm soát được các luồng xử lý và tạo ra các thành phần xử lý nghiệp vụ chuyên biệt hóa.
* Tạo thành mô hình chuẩn cho nhiều dự án, các chuyên gia sẽ tiếp cận – tìm hiểu những dự án đó một cách nhanh chóng và hiệu quả.
* Giúp các chuyên gia lập trình, nhà quản lý, nhà đầu tư, PM… có thể hiểu được dự án hoạt động ra sao hoặc giúp các lập trình viên dễ dàng quản lý – phát triển dự án.
* Đây là một mô hình chuẩn, nó tối ưu nhất hiện nay so với nhiều mô hình khác và được sử dụng trong nhiều dự án và nhiều lĩnh vực, đặc biệt trong công nghệ sản xuất ứng dụng – phần mềm. Các lập trình viên sử dụng mô hình chuẩn MVC để có thể dễ dàng phân phối và chuyển giao công nghệ.
* Đây là mô hình đơn giản, xử lý những nghiệp vụ đơn giản, và dễ dàng triển khai với các dự án nhỏ.

Nhược điểm:

* Yêu cầu về chuyên môn khá.
* Có kiến thức vững về các mô hình chuẩn.
* Khó triển khai với những dự án yêu cầu phúc tạp hơn.
* Hiện nay đang có một khái niệm mô hình mới đó là HMVC đang dần thay thế cho MVC.

## Spring Framework

### Đôi nét về lịch sử

#### EJB và Spring

Có thể nói rằng Spring Framework ra đời là nhờ sự thành công của EJB. EJB (Enterprise JavaBeans) là một kiến trúc phát triển dành cho các ứng dụng doanh nghiệp đòi hỏi tính mạnh mẽ và khả năng mở rộng cao. EJB là một thành phần quan trọng của nền tảng Java EE. EJB được tạo ra bởi hai gã khổng lồ công nghệ là IBM và Sun vào những năm cuối thập niên 90. Dưới sự phát triển của hai tập đoàn này, EJB đã trở nên nặng nề hơn. Tiêu chuẩn EJB cũng nhanh chóng được thông qua. Sau tất cả, EJB được ca ngợi là giải pháp tốt nhất để xây dựng các ứng dụng phân tán dành cho doanh nghiệp.

Nhưng rồi những chỉ trích về chi tiết kỹ thuật của EJB đã tăng nhanh còn hơn sự nổi tiếng của nó. EJB là một framework lớn với độ phức tạp cao. Các nhà phát triển khi làm việc với EJB đã nhận ra rằng nó không hoạt động như những gì nó đã đề ra. EJB chỉ tuyệt vời trên bàn giấy, nhưng ra thực tế đó lại là một chuyện khác.

EJB vẫn được tiếp tục phát triển, dễ sử dụng hơn nhiều so với buổi ban đầu. Nhưng những chỉ trích vẫn còn đó, nỗi thất vọng trong cộng đồng Java ngày càng tăng lên, đòi hỏi cần có một giải pháp thay thế.

#### Thuận lợi của EJB

* Transaction Management
* Declarative Transaction support
* Persistence (CMP & BMP)
* Declarative Security
* Distributed Computing (Container managed RPC)

#### Sự ra đời của Spring

Vào năm 2002, Spring Framework phát hành phiên bản đầu tiên bởi Rod Johnson. Việc xây dựng các ứng dụng doanh nghiệp trở nên đơn giản và dễ dàng hơn. Rod Johnson đã cho xuất bản một cuốn sách được phổ biến rộng rãi có nhan đề "J2EE Development without EJB". Điều này đã cho Spring Framework rất nhiều động lực để thay thế EJB.

Theo đà phát triển đó, Spring đã trở thành framework mã nguồn mở phổ biến nhất để xây dựng các ứng dụng doanh nghiệp. Cách tiếp cận thực tế ban đầu của Rod Johnson tiếp tục được phát triển và hướng tới một bộ công cụ hoàn chỉnh dành cho xây dựng các ứng dụng doanh nghiệp. Theo một số nguồn, trên 50% các ứng dụng web Java hiện nay đang sử dụng Spring.

Để ngăn chặn sự phức tạp trong phát triển các ứng dụng, Spring Framework thường dựa trên các quan điểm như sau:

* Đơn giản hóa công việc phát triển thông qua việc sử dụng các POJO (Plain Old Java Object)
* Nới lỏng ràng buộc giữa các thành phần thông qua việc sử dụng Dependency Injection
* Giảm thiểu các mã boilerplate thông qua việc sử dụng template và aspect ...

#### Thuận lợi của Spring khi so sánh với EJB

* Testing dễ dàng hơn – không cần khởi động EJB container để test.
* Spring dựa vào quy ước của chuẩn Java Bean, nên programmer dễ dàng làm việc với nó.
* Nó sử dụng AOP (Aspect-Oriented Programming), mô hình hữu ích để bổ sung vào OOP truyền thống và bảo toàn tính nguyên vẹn của OOP.
* Uyển chuyển.

Mục đích của Spring là trở thành một application framework. Các framework phổ biến khác như Struts, Tapestry, JSF, … Là các framework tốt cho tầng web nhưng khi chúng ta sử dụng các framework này, chúng ta phải cung cấp thêm framework khác để giải quyết tầng enterprise mà tích hợp tốt với các framework này. Spring làm giảm bớt vấn đề này bằng cách cung cấp một framework toàn diện bao gồm:

* Core bean container.
* MVC framework.
* AOP integration framework.
* JDBC integration framework.
* EJB integration framework.

### Một số khái niệm chính

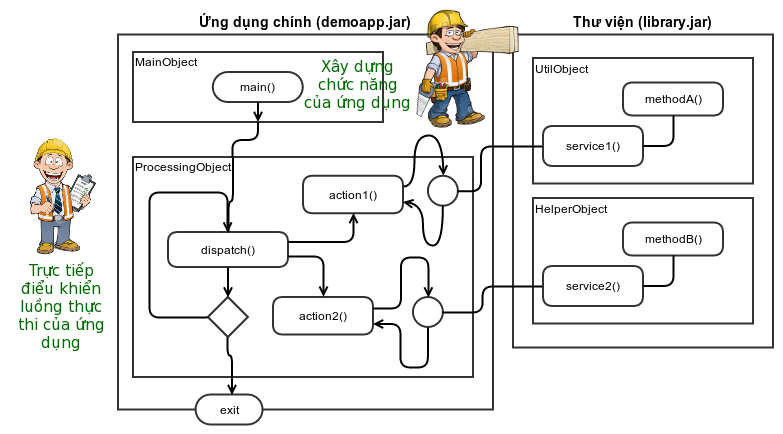
Phần này sẽ trình bày 2 khái niệm chính, thứ làm nên sức mạnh của Spring là Ioc và DI.

#### Inversion of Control (IoC)

IoC Container trong Spring được xây dựng dựa trên nguyên lý Inversion of Control (đảo ngược điều khiển) đã xuất hiện khá lâu trong các mẫu hình thiết kế (design pattern), và được phổ biến rộng rãi nhờ Robert C. Martin và Martin Fowler. Để hiểu về Spring trước tiên chúng ta cần hiểu khái niệm IoC là gì? Muốn vậy, chúng ta cần trả lời câu hỏi: Control (điều khiển) trong chương trình phần mềm là gì, và Inversion (sự đảo ngược) đối với điều khiển trong ngữ cảnh này được hiểu như thế nào?

Khái niệm Control Flow (tạm dịch là luồng thực thi) được sử dụng cho trình tự thực hiện các câu lệnh, chỉ thị hoặc lời gọi hàm trong một chương trình, khi chương trình này thực thi.

Do chương trình ngày càng phức tạp, nên các lập trình viên áp dụng phương pháp lập trình hướng đối tượng nhằm phân loại, chia tách các chức năng và gom thành các đối tượng. Các lập trình viên còn tạo dựng các thư viện tạo sẵn để có thể sử dụng lại. Luồng thực thi của chương trình, trong những tình huống cần xem xét ở mức tổng thể, không còn quan tâm đến các bước thực thi câu lệnh cụ thể nữa, mà chỉ xem xét đến quá trình gọi phương thức của các đối tượng trong ứng dụng cũng như các đối tượng của thư viện dựng sẵn.



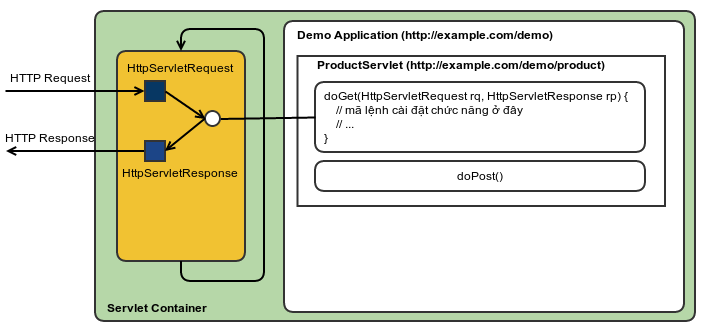
Hình 3‑5 Luồng của một chương trình bình thường

Như ví dụ trên, khi chúng ta xây dựng ứng dụng từ đầu, đã thực hiện hai nhiệm vụ: trực tiếp điều khiển luồng thực thi của chương trình và xây dựng các chức năng để đáp ứng nghiệp vụ của ứng dụng. Thực tế, có nhiều chương trình hoặc bộ phận trong chương trình có luồng thực thi rất giống nhau, chẳng hạn phần tương tác với HTTP trong các ứng dụng web, phần unit testing trong các ứng dụng, … Việc trực tiếp tạo dựng và kiểm soát luồng thực thi của chương trình lặp đi lặp lại khi xây dựng nhiều ứng dụng sẽ làm mất nhiều công sức, chi phí, tạo ra sự nhàm chán và dễ phát sinh lỗi. Điều này tạo ra động lực cũng như môi trường để nguyên lý đảo ngược điều khiển nảy nở và phát triển.

Vậy sự “đảo ngược” (inversion) luồng điều khiển chương trình trong ngữ cảnh này được hiểu như thế nào? Chúng ta hãy xem xét 2 ví dụ dưới đây.

Ví dụ 1: Ứng dụng web trong Java với Java Servlet

Hầu như tất cả chúng ta khi lập trình ứng dụng web bằng Java Servlet đều hình dung rõ ràng các bước lập trình như sau: Tạo lớp đối tượng kế thừa từ HttpServlet, nạp đè các phương thức doGet() - doPost(), sau đó đăng ký trong file cấu hình Deployment Descriptor tương ứng Servlet này với đường dẫn xác định. Lớp đối tượng Servlet chúng ta tạo ra sẽ được gọi đến khi có một truy vấn HTTP có đường dẫn “khớp” với đường dẫn khai báo trong Deployment Descriptor.

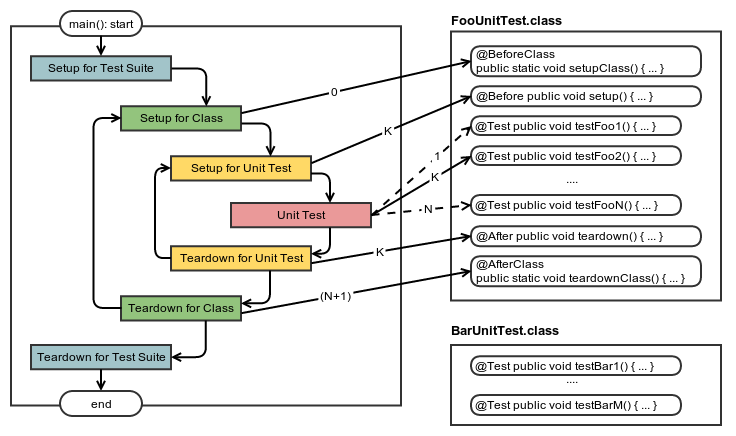


Hình 3‑6 Servlet Container

Vậy ai kích hoạt ứng dụng chúng ta viết để đáp ứng mỗi khi có Http Request gửi đến? ai chịu trách nhiệm chuyển đổi các thông điệp Http (HTTP Request và HTTP response) thành các đối tượng Java (HttpServletRequest và HttpServletResponse) để truyền cho các hàm doGet(), doPost()? chính là Servlet Container.

Ví dụ 2: Lập trình Unit Testing bằng Junit

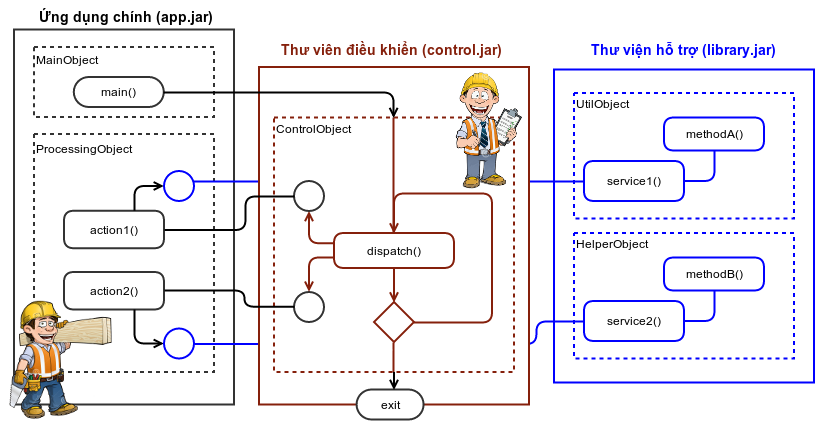
Trong quá trình phát triển các thành phần chức năng của ứng dụng, chúng ta thường áp dụng kiểm thử thành phần (Unit Testing) để đảm bảo chức năng đó vẫn chạy đúng trong suốt quá trình ứng dụng được mở rộng và phát triển thêm. Để tạo bộ Unit Test, chúng ta chỉ cần tạo một lớp đối tượng, định nghĩa các phương thức khởi tạo, phương thức kết thúc và các phương thức test. Sau đó, chúng ta chỉ việc chạy bộ test để kiểm thử.



Hình 3‑7 UnitTest

Việc điều khiển trình tự thực thi các phương thức được giao cho thư viện bên ngoài đảm nhiệm (chẳng hạn như TestNG hoặc JUnit).

Với hai ví dụ trên, chúng ta nhận thấy trong các ứng dụng đã có sự thay đổi vai trò: ứng dụng không còn vừa trực tiếp tạo dựng và kiểm soát luồng thực thi, vừa xây dựng chức năng nghiệp vụ. Việc kiểm soát luồng thực thi được tách khỏi chức năng nghiệp vụ và bị đẩy ra bên ngoài.



Hình 3‑8 Luồng của một chương trình IoC

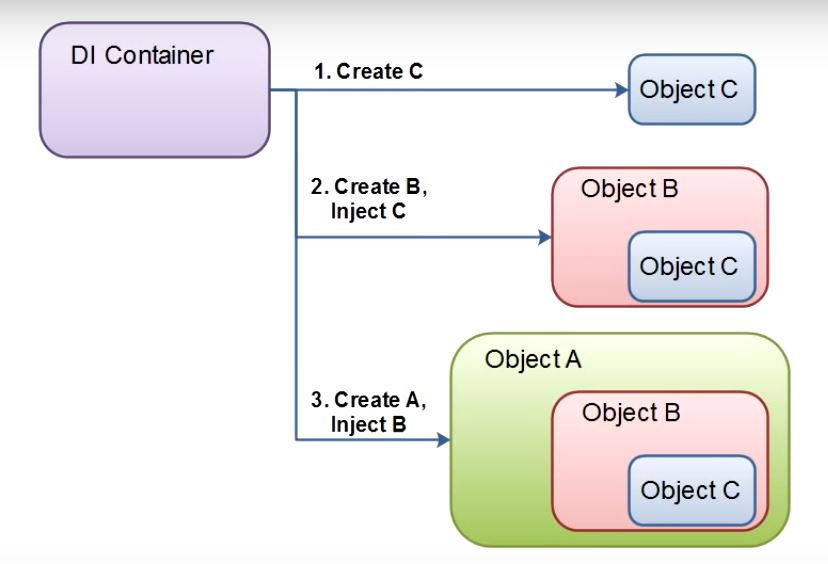
Như vậy, khái niệm “đảo ngược” ở đây chính là chuyển nhiệm vụ kiểm soát lưu trình thực thi từ ứng dụng cho một thành phần chuyên trách (thường là một thư viện phần mềm khung – framework – dựng sẵn ở bên ngoài). Ứng dụng chính chúng ta quan tâm phát triển không kiểm soát việc điều khiển luồng thực thi nữa, mà chỉ tập trung vào việc định nghĩa chức năng. Thư viện phần mềm khung chuyên trách kiểm soát điều khiển sẽ dựa trên mô tả trong cấu hình của ứng dụng để thay mặt ứng dụng điều phối luồng thực thi trong chương trình.

Dễ dàng nhận thấy thư viện phần mềm khung này khác với các thư viện thông thường ở chỗ: thư viện thông thường cung cấp các chức năng và chờ được ứng dụng gọi đến, còn thư viện phần mềm khung tạo dựng luồng thực thi và gọi đến các chức năng của ứng dụng.

Nói một cách ngắn gọn IoC là một design pattern và tập hợp các kỹ thuật lập trình liên quan, trong đó luồng thực thi của một hệ thống bị đảo ngược so với cách tương tác truyền thống. IoC trong Spring cũng hoàn toàn mang ý nghĩa như trên.

#### Dependency Injection (DI)

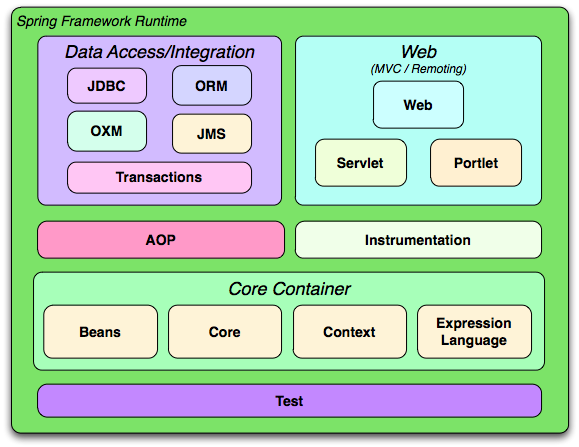
Dependency Injection là khả năng liên kết giữa các thành phần lại với nhau, đó chính là việc các thuộc tính trong một đối tượng được “tiêm chích” (injection) để tham chiếu lần lượt đến các đối tượng khác. Dependency Injection trong Spring hoạt động dựa trên Java Reflection như đã trình bày trong phần [3.1].



Hình 3‑9 Dependency Injection

### Các Module

Spring framework có thể được xem như một kiến trúc chứa 7 module. Chức năng của mỗi thành phần như sau:



Hình 3‑10 Kiến trúc Spring Framework

#### Core Container:

Core container cung cấp chức năng cơ bản của Spring. Thành phần chính của nó là Bean Factory, một cài đặt của Factory pattern. BeanFactory áp dụng IoC pattern để đặc tả sự phụ thuộc từ code của ứng dụng.

#### Spring Context/Application Context:

Spring context là một file cấu hình để cung cấp thông tin ngữ cảnh của Spring. Spring context cung cấp các service như JNDI access, EJB integration, e-mail, internalization, validation, và scheduling functionality.

#### Spring AOP (Aspect-Oriented):

Spring AOP module tích hợp chức năng lập trình hướng khía cạnh vào Spring framework thông qua cấu hình của nó. Spring AOP module cung cấp các dịch vụ quản lý giao dịch cho các đối tượng trong bất kỳ ứng dụng nào sử dụng Spring. Với Spring AOP chúng ta có thể tích hợp declarative transaction management vào trong ứng dụng mà không cần dựa vào EJB component.

Spring AOP module cũng đưa lập trình metadata vào trong Spring. Sử dụng cái này chúng ta có thể thêm annotation vào source code để hướng dẫn Spring nơi và làm thế nào để liên hệ với aspect.

#### Spring DAO:

Tầng JDBC và DAO đưa ra một cây phân cấp exception để quản lý kết nối đến database, điều khiển exception và thông báo lỗi được ném bởi vendor của database. Tầng exception đơn giản điều khiển lỗi và giảm khối lượng code mà chúng ta cần viết như mở và đóng kết nối. Module này cũng cung cấp các dịch vụ quản lý giao dịch cho các đối tượng trong ứng dụng Spring.

#### Spring ORM:

Spring có thể tích hợp với một vài ORM framework để cung cấp Object Relation tool bao gồm: JDO, Hibernate, OJB và iBatis SQL Maps.

#### Spring Web module:

Nằm trên application context module, cung cấp context cho các ứng dụng web. Spring cũng hỗ trợ tích hợp với Struts, JSF và Webwork. Web module cũng làm giảm bớt các công việc điều khiển nhiều request và gắn các tham số của request vào các đối tượng domain.

#### Spring MVC Framework:

MVC Framework thì cài đặt đầy đủ đặc tính của MVC pattern để xây dựng các ứng dụng Web. MVC framework thì cấu hình thông qua giao diện và chứa được một số kỹ thuật view bao gồm: JSP, Velocity, Tiles và generation of PDF và Excel file.

### Spring Core

#### Bean

##### Tổng quan về bean

Trong Spring, các đối tượng tạo thành xương sống cho ứng dụng của được quản lý bởi bộ chứa Spring IoC được gọi là bean. Bean là một đối tượng được khởi tạo, lắp ráp và được quản lý bởi bộ chứa Spring IoC. Mặt khác, bean chỉ đơn giản là một trong nhiều đối tượng trong ứng dụng. Bean và các phụ thuộc giữa chúng, được phản ánh trong siêu dữ liệu cấu hình được sử dụng bởi một container.

Một Spring container sẽ quản lý một hoặc nhiều bean. Các bean được tạo ra nhờ vào thông tin cấu hình được chỉ ra trong thẻ <bean /> nếu sử dụng XML config. Bên trong container các bean được biểu diễn như một đối tượng của lớp BeanDefinition, nó chứa các thông tin sau:

* Class của đối tượng mà bean được cấu hình (bao gồm cả phần package)
* Các thông số cấu hình hành vi của bean như: phạm vi (scope), vòng đời (lifecycle)…
* Các tham chiếu đến các bean khác, các tham chiếu này được gọi là collaborators hoặc dependencies
* Các thiết lập khác như số connection kết nối đến bean…
* Các thuộc tính sau đây thường được sử dụng để định nghĩa một bean: class (instantiating beans), name (naming beans), scope, constructor arguments, properties, autowiring mode, lazy-initialization mode, initialization method, destruction method.

##### Scope của bean

Scope của một bean định nghĩa vòng đời của bean đó trong Context. Spring Framework định nghĩa 6 types của scopes như sau:

* Singleton
* Prototype
* Request
* Session
* Application
* Websocket

4 Scope cuối bao gồm Request, Session, Application, Websocket chỉ dùng cho Web Application.

###### Singleton Scope

Singleton là scope mặc định của Spring bean, Spring container sẽ tạo một sự thể hiện cho bean đó sau đó lưu vào bộ nhớ cached và tất cả các request tới bean đó sẽ trả về duy nhất sự thể hiện này. Dưới đây là một ví dụ về tạo bean với scope là singleton bằng cách sử dụng @Scope annotation:

**public** **class** Customer {

**private** **int** id;

**private** String name;

**private** String address;

@Bean @Scope("singleton")

**public** Customer customerSingleton(){

**return** **new** Customer();

}

}

Chúng ta có thể sử dụng hằng số thay vì sử dụng String Value như sau: ConfigurableBeanFactory.SCOPE\_SINGLETON.

Chúng ta cũng có thể định nghĩa singleton bean bằng XML configuration như dưới đây:

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"* scope=*"singleton"*/>

###### Prototype Scope

Một bean được định nghĩa với scope là prototype thì Spring container sẽ trả về sự thể hiện khác nhau cho mỗi request. Sử dụng annotation @Scope để định nghĩa một bean có scope là protory như sau:

@Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

Chúng ta cũng có thể định nghĩa prototype bean bằng XML configuration bằng cách khai báo giá trị cho thuộc tính scope=“prototype”

###### Request scope

Một bean được định nghĩa là request scope có nghĩa là Spring container sẽ tạo một sự thể hiện cho một HTTP request.

Sử dụng @Scope annotation để định nghĩa một bean có scope là request như sau:

@Scope(value = WebApplicationContext.SCOPE\_REQUEST,

proxyMode = ScopedProxyMode.TARGET\_CLASS)

proxyMode = ScopedProxyMode.TARGET\_CLASS được dùng để tạo AOP proxy của bean. Chúng ta khai báo giá trị này khi muốn inject một bean có scope request, session, application vào một bean có scope singleton, prototype. Hoặc inject một bean có scope prototype vào bean có scope singleton.

###### Session scope

Một bean được định nghĩa là session scope có nghĩa là Spring container sẽ tạo một sự thể hiện cho một HTTP session. Sử dụng @Scope để định nghĩa một session bean như sau:

@Scope(value = WebApplicationContext.SCOPE\_SESSION,

proxyMode = ScopedProxyMode.TARGET\_CLASS)

###### Application scope

Application scope sẽ tạo một sự thể hiện cho một vòng đời của một ServletContext.

Application scope thì cũng giống như singleton-scope tuy nhiên có sự khác biệt quan trọng là scope của application-scope sẽ được chia sẽ giữa các servlet-based chạy trên cùng một ServletContext trong khi singleton-scope thì scoped cho một application context.

Tạo một bean application-scope như sau:

@Scope(value = WebApplicationContext.SCOPE\_APPLICATION,

proxyMode = ScopedProxyMode.TARGET\_CLASS)

###### WebSocket Scope

WebSocket-scope bean được khởi tạo và lưu trữ tại WebSocket session attributes. Sự thể hiện trả về là giống nhau khi có request từ WebSocket session.

@Scope(value = "websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS)

#### Spring IoC Container

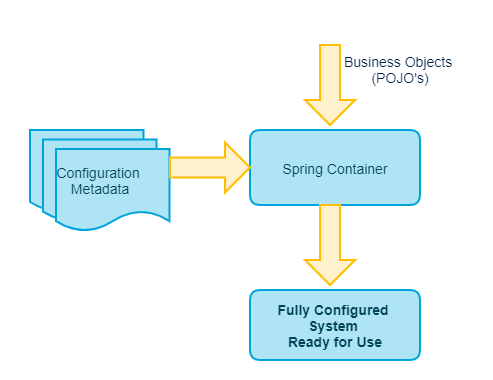
Spring là một framework thực thi theo nguyên tắc Inversion of Control, IoC cũng được biết đến như Dependency Injection. Nó là một quá trình xử lý ở nơi các object định nghĩa sự phụ thuộc (dependency). Khi các đối tượng hoạt động với nhau chỉ thông qua các tham số của constructor, tham số của các method hoặc các property để thiết lập thể hiện sau khi được khởi tạo. Container sau đó “tiêm” các đối tượng phụ thuộc khi nó được tạo ra từ các bean. Quá trình này về cơ bản là sự đảo ngược, các bean sẽ điều khiển các thể hiện hoặc vị trí phụ thuộc bằng cách khởi tạo trực tiếp từ class của chúng.

Package org.springframework.beans và org.springframework.context là 2 package cơ bản cho IoC container của Spring. Interface BeanFactory cung cấp kỹ thuật để cấu hình nâng cao và quản lý bất kỳ loại object nào. ApplicationContext kế thừa BeanFactory, ApplicationContext thêm vào một số tính năng như tích hợp để dễ dàng hoạt động với các tính năng của Spring AOP (Aspect Oriented Programming) như Message resource handling, event publication và một số lớp context đặc biệt như WebApplicationContext được sử dụng trong các ứng dụng web. Nói một cách ngắn gọn, BeanFactory cung cấp cơ chế cấu hình cho framework và các chức năng cơ bản. ApplicationContext thêm vào một số chức năng nâng cao. Trong phần này sẽ sử dụng ApplicationContext để để mô tả cho IoC container của Spring.

Interface org.springframework.context.ApplicationContext chịu trách nhiệm khởi tạo, cấu hình và phân phát các bean. Container lấy các chỉ dẫn cho việc khởi tạo, cấu hình và phân phát bằng cách đọc các configuration metadata. Các configuration metadata được trình bày bằng nhiều cách khác nhau như trong file XML, Java annotation hoặc trong Java code. Điều này cho phép biểu diễn các đối tượng và sự phụ thuộc lẫn nhau giữa chúng trở nên phong phú hơn.

Một số class thực thi (implementation) interface ApplicationContext như Class PathXmlApplicationContext, FileSystemXmlApplicationContext, AnnotationConfigApplicationContext, … XML là cách truyền thống và thường được sử dụng để định nghĩa các configuration metadata, chúng ta cũng có thể khai báo để container sử dụng các annotation như configuration metadata hoặc trực tiếp trong code Java bằng cách cấu hình thêm trong file XML cho phép sử dụng các annotation như sau: <context:annotation-config/>

Hình 3-11 mô tả cách thức làm việc của Spring. Các class trong ứng dụng sẽ kết hợp với các configuration metadata để sau khi ApplicationContext được tạo ra và khởi tạo chúng ta có được một cấu hình đầy đủ và có thể thực thi được.



Hình 3‑11 Spring IoC Container

Thông thường các configuration metadata được biểu diễn dưới dạng XML, vì vậy trong phần này sẽ sử dụng configuration metadata dưới dạng này để nói về các khái niệm và tính năng của IoC container.

Lưu ý rằng không phải chỉ có các configuration metadata dưới dạng XML mới được Spring container chấp nhận, như đã nói ở phần trên ngoài các metadata dạng XML chúng ta còn có thể sử dụng Java annotaion hoặc trực tiếp trong code Java (Java config). Kể từ phiên bản 2.5 Spring đã hổ trợ kiểu cấu hình dựa trên các Java annotation. Và kể từ Spring 3.0 nhiều tính năng đã được cung cấp bằng cách sử dụng Java config và nó trở thành một phần của Spring core. Vì vậy chúng ta có thể định nghĩa bean bên ngoài các class của ứng dụng bằng cách sử dụng Java code. Để sử dụng tính năng này xem thêm các annotation như @Configuration, @Bean, @Import và @DependOn.

Cấu hình trong Spring bao gồm ít nhất là một và thường là nhiều hơn một định nghĩa bean mà container cần phải quản lý. Trong metadata dạng XML để khai báo và cấu hình một bean ta sử dụng thẻ <bean /> bên trong thẻ <beans />. Nếu sử dụng dạng Java config thông thường sẽ sử dụng annotaion @Bean bên trong một class với annotaion @Configuration. Ví dụ dưới đây là sẽ cho thấy cấu trúc cơ bản của một file XML trình bày các configuration metadata.

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"customer1"* class=*"stu.ytbhoanghai.abc.Customer"*>

<property name=*"id"* value=*"18"*/>

</bean>

<bean id=*"customer2"* class=*"stu.ytbhoanghai.abc.Customer"*>

<property name=*"id"* value=*"19"*/>

</bean>

</beans>

Thuộc tính id chỉ ra định danh của bean đó, thuộc tính class chỉ ra class của bean này. Giá trị của thuộc tính id sẽ được các bean khác sử dụng nếu cần thiết.

##### Khởi tạo container

Việc khởi tạo một Spring container đơn giản như việc chúng ta khởi tạo một đối tượng, cần chỉ rõ đường dẫn của file XML cấu hình trong khi khởi tạo một container. Ví dụ dưới đây sẽ cho chúng ta thấy rõ hơn:

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-context.xml");

##### Sử dụng container

Ví dụ dưới đây sẽ chỉ ra cách đọc các định nghĩa của bean và truy cập đến chúng.

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-context.xml");

Customer customer = (Customer) context.getBean(Customer.**class**);

customer.setName("Hoang Hai");

customer.setAddress("180 Cao Lo");

System.***out***.println(customer);

((ClassPathXmlApplicationContext) context).close();

}

#### Dependency Injection

##### Constructor – based dependency injection

Constructor – based DI là phương pháp “tiêm sự phuộc” cho các đối tượng bằng cách gọi hàm tạo (constructor method) với các tham số chính là các “sự phụ thuộc” (dependentcies) của đối tượng. Ví dụ sau đây sẽ đưa ra một class có thể tiêm sự phụ thuộc bằng phương pháp này.

**public** **class** Oder {

**private** String data;

**public** Oder(String data) {

**super**();

**this**.data = data;

}

**public** String getData() {

**return** data;

}

**public** **void** setData(String data) {

**this**.data = data;

}

@Override

**public** String toString() {

**return** "Oder [data=" + data + "]";

}

}

Khi đó để tiêm sự phụ thuộc cho lớp Oder cần khai báo bean như sau:

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/>

</bean>

Thẻ <constructor-arg /> trong ví dụ này sẽ cho Spring container biết đây là một bean mà các dependentcy sẽ được tiêm vào bằng cách gọi hàm tạo của lớp này. Đối với các hàm tạo với nhiều đối số thì thứ tự của các thẻ <constructor-arg /> sẽ giống với thứ tự các đối số của hàm, nhưng chúng ta cũng có thể thay đổi điều này bằng cách chỉ định giá trị cho biến type là tham chiếu đến class sẽ là sự thể hiện của đối số.

Ở ví dụ trên các dependency là những kiểu dữ liệu nguyên thủy nên thẻ <constructor-arg /> sử dụng thuộc tính value để tham chiếu. Riêng đối với các kiểu dữ liệu do người dùng tự định nghĩa (extend Object) thì khai báo bean như sau:

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"address"* ref=*"address1"*/>

</bean>

address1 là một bean được khai báo trong Spring Container, là một sự thể hiện của một class có tên là Address.

##### Setter – based dependency injection

Mỗi bean trong Spring Container phải hội đủ tiêu chuẩn của một đối tượng Java Bean, có nghĩa là sẽ có các phương thức getXXX() và setXXX() cho các thuộc tính được khai báo trong class đó. Spring cũng cung cấp cho chúng ta phương pháp để gán giá trị đến một thuộc tính qua phương thức setter của nó.

Setter – based DI là phương pháp tiêm sự phuộc cho các đối tượng bằng cách gọi các setter của một class sau khi hàm dựng mặc định (không đối số) được gọi để khởi tạo bean. Ví dụ dưới đây sẽ đưa ra một class có thể tiêm sự phụ thuộc bằng các setter.

**public** **class** Customer {

**private** **int** id;

**private** String name;

**private** Oder oder;

**public** Oder getOder() {

**return** oder;

}

**public** **void** setOder(Oder oder) {

**this**.oder = oder;

}

**public** Customer() {

**super**();

**this**.name = "Hoang Hai";

}

@Override

**public** String toString() {

**return** "Customer [id=" + id

+ ", name=" + name

+ ", oder=" + oder + "]";

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

Đơn giản là khai báo các phương thức setter cho các thuộc tính trong class. Khi đó, để tiêm sự phụ thuộc cho class Customer, ta khai báo như sau:

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/>

</bean>

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"*>

<property name=*"oder"* ref=*"oder"*/>

</bean>

</beans>

Thẻ <property> được dùng để biểu diễn cho một phương thức setter. Thuộc tính name của thẻ <property /> chỉ định thuộc tính mà setter của nó sẽ được gọi để gán giá trị được chỉ định trong thuộc tính ref.

Chúng ta có thể kết hợp cả 2 phương pháp DI này trong khai báo một bean. Thông thường Constructor – based DI được sử dụng cho những dependency bắt buộc và Setter – based DI cho những dependency tùy chọn.

Dễ dàng nhận thấy rằng tất cả các class ở những ví dụ trên đều là những POJOs (những class Java bình thường), chúng hoàn toàn không kế thừa từ bất cứ một class hoặc thực thi một interface đặc biệt nào, đều này cho thấy được sức mạnh của Spring trong việc xây dựng một ứng dụng từ các POJOs là nhờ vào DI và IoC container.

Spring container xử lý việc tiêm sự phụ thuộc cho các bean như sau:

* Spring container ApplicationContext được tạo ra và khởi tạo những siêu dữ liệu cấu hình (configuration metadata) cho tất cả các bean. Các configuration metadata có thể là XML, code Java hoặc các annotaion.
* Đối với từng bean các dependency của nó được thể hiện dưới dạng là các property, các đối số của hàm tạo. Những dependency này sẽ phải được cung cấp cho bean khi bean thực sự được khởi tạo.
* Đối với từng property hoặc các đối số của hàm được gán giá trị để thiết lập cho bean hoặc được tham chiếu đến một bean khác trong container.
* Giá trị của từng property và đối số của hàm tạo sẽ được chuyển thành kiểu dữ liệu phù hợp, Spring sẽ chuyển chúng từ kiểu String sang các kiểu dữ liệu khác như int, long, String, boolean…

##### Autowiring

Khi một bean A cần tham chiếu đến bean B (B là một dependency của A) chúng ta cần phải cấu hình cho bean A để có thể “tiêm” B vào thông qua Constructor – based DI hoặc Setter – based DI. Tuy nhiên nếu chúng ta cấu hình cho bean A autowire thì Spring container có thể thực hiện việc này một cách “tự động” bằng cách Spring container sẽ kiểm tra trong ApplicationContext để tìm ra B và “tiêm” cho A.

Cũng từ ví dụ bean customer và oder từ phía trên, ta chỉnh sửa mã nguồn một chút để thấy rõ sự khác biệt mà Autowiring mang lại.

Trong class Customer chúng ta sẽ thêm annotaion vào thuộc tính oder như sau:

**public** **class** Customer {

**private** **int** id;

**private** String name;

@Autowired **private** Oder oder;

**public** Customer() {

**super**();

**this**.name = "Hoang Hai";

}

// get and set

}

Bean customer được khai báo như sau:

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"*/>

Rõ ràng chúng ta không hề inject oder vào bean customer theo bất kì cách nào, khi chạy sẽ cho ra kết quả như sau:

**private** **static** Customer *customer*;

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

ApplicationContext context = **new**

ClassPathXmlApplicationContext("spring-context.xml");

*customer* = (Customer) context.getBean("customer");

*customer*.setName("Hoang Hai");

System.***out***.println(*customer*);

((ClassPathXmlApplicationContext) context).close();

}

Kết quả:

Customer [id=0, name=Hoang Hai, oder=Oder [data=This is a Data]]

Autowiring thực sự hữu ích khi phát triển một ứng dụng mà việc chỉnh sửa mã nguồn là thường xuyên, ví dụ khi thêm một thuộc tính mới cho một class, nếu sử dụng autowire thì cấu hình bean của class này không cần phải thay đổi.

Autowire có 4 loại vì vậy cần phải chỉ rõ loại của autowire đối với mỗi bean. Sau đây là 4 loại của autowire:

* Auto-wiring “no”: Đây là cách mà chúng ta áp dụng trong ví dụ ở trên. Nó cũng là chế độ auto-wiring mặc định. Các bean tham chiếu đến nhau thông qua thuộc tính ref. Auto-wiring ‘no’ được khuyến cáo nên sử dụng, bởi vì việc chỉ định các dependency một cách rõ ràng sẽ tốt hơn và ở một mức độ nào đó nó được xem như tài liệu cho cấu trúc của hệ thống.
* Auto-wiring “byName”: Trong trường hợp này, Spring container sẽ tìm bean có id trùng với tên của property cần autowire và thực thiện auto wired thông qua method setter.

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/></bean>

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"*

autowire=*"byName"*/>

Hãy chú ý đến bean customer có thuộc tính autowire được chỉ định là “byName” String container sẽ tự động tìm bean có id là oder để tiêm và thuộc tính oder thông qua phương thức setter trong đối tượng Customer.

* Auto-wiring “byType”: Autowire by type cho phép một property được autowire nếu có một bean có kiểu (class) là kiểu của property này. Nếu có nhiều hơn một bean thì một exception được ném ra, ngược lại nếu tồn tại bất cứ bean nào phù hợp thì không có exception nào được ném ra. Nó sẽ tiêm thông qua phương thức setter.

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/></bean>

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"*

autowire=*"byType"*/>

Chúng ta có thể thấy bean oder có kiểu class Oder trùng với kiểu mà bean customer yêu cầu, vì vậy chính nó sẽ được “tiêm” vào bean customer.

* Auto-Wiring “constructor”: Tương tự như autowire by type, tuy nhiên autowire by constructor chỉ áp dụng cho các đối số của hàm tạo. Nếu không có chính xác một bean có kiểu là kiểu của các đối số trong hàm tạo thì một exception sẽ được ném ra.

Thêm phương thức khởi dựng cho class Customer như sau:

**public** Customer(Oder oder) {

**super**();

**this**.oder = oder;

}

Thay đổi thuộc tính autowire của bean customer như sau:

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"* type=*"java.lang.String"*/></bean>

<bean id=*"customer"* class=*"stu.ytbhoanghai.abc.Customer"*

autowire=*"constructor"*/>

Thay vì “tiêm” bằng phương thức setter, Spring sẽ “tiêm” bean oder vào bean customer bằng constructor như trên.

#### Tìm hiểu về annotation trong Spring Framework

Chúng ta không thể phủ nhận tính linh hoạt, mạnh mẽ và đơn giản của các annotation trong việc định nghĩa các thông tin cấu hình trong Spring Framework, sau đây sẽ trình bày một số các annotation đi kèm ví dụ mà chúng ta thường sử dụng khi làm việc với Spring Framework.

Có lẽ cá nhân đọc phần này sẽ thấy “kỳ lạ” khi ở một trong những phần trên, chúng ta đã đề cập rằng sẽ sử dụng configuration metadata XML để khởi tạo container mà cụ thể là phần [3.3.4.2] Spring IoC Container. Thì giờ đây lại đi tìm hiểu về các annotation. Chúng ta phải biết rằng Spring cho phép kết hợp cả 3 cách (java code, annotation, xml) để cấu hình cũng như khởi tạo các bean. Vì vậy việc sử dụng cách nào hay kết hợp như thế nào là tùy thuộc vào cá nhân người lập trình sao cho thuận tiện, dễ hiểu, dễ chỉnh sửa cũng như nâng cấp mã...

##### @Configuration

Chúng ta đã biết cách khởi tạo bean trong Spring Container bằng cách sử dụng tập tin XML trong các ví dụ trên. Ngoài cách đó ra Spring cũng hỗ trợ chúng ta khởi tạo các bean bằng cách sử dụng annotation @Configuration. Để làm được điều này, chúng ta đặt annotation trước class định nghĩa các bean sẽ được tạo như sau:

@Configuration

**public** **class** ApplicationConfiguration {

@Bean

**public** Customer customer() {

**return** **new** Customer();

}

@Bean

**public** Oder oder() {

**return** **new** Oder("This is Data");

}

}

Class trên định nghĩa 2 bean sẽ được tạo trong Spring Container là customer và oder, để có thể lấy ra được 2 bean trên chúng ta sẽ sử dụng một Implementation khác của ApplicationContext là AnnotationConfigApplicationContext như sau:

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

ApplicationContext context = **new**

AnnotationConfigApplicationContext

(ApplicationConfiguration.**class**);

*customer* = (Customer) context.getBean("customer");

*customer*.setName("Hoang Hai");

System.***out***.println(*customer*);

((AnnotationConfigApplicationContext)context).close();

}

Kết quả:

Customer [id=0, name=Hoang Hai, oder=Oder [data=This is Data]]

##### Auto component scan

Cũng tương tự như cách khởi tạo bean bằng annotation @Configuration đã trình bày ở ví dụ trên, ở cách này Spring sẽ scan qua các class mà chúng ta đặt các annotation để định nghĩa bean. Cách này thường được sử dụng nhiều hơn bởi các lập trình viên vì nó cung cấp các annotation (@Component, @Repository, @Service, @Controller) với tên phù hợp với nhiệm vụ của từng bean mà chúng ta muốn định nghĩa.

* @Component: Dùng cho những đối tượng không liên quan để database, business logic hay persentation layer.
* @Repository: Dùng cho những đối tượng liên quan đến database layer.
* @Service: Dùng cho những đối tượng liên quan đến Business logic layer.
* @Controller: Dùng cho những đối tượng liên quan đến persentation layer.

Thật ra không có quy định cụ thể cho việc sử dụng các annotation trên cho bất cứ các đối tượng nào, nhưng để làm rõ hơn về mặt ngữ nghĩa cho từng đối tượng chúng ta nên sử dụng các annotation trên cho phù hợp với mục đích.

@Component

**public** **class** Customer {

**private** **int** id;

**private** String name;

@Autowired **private** Oder oder;

**public** Customer() {

**super**();

**this**.name = "Hoang Hai";

}

//get and set

}

Hãy chú ý rằng chúng ta có annotation @Component được đặt trên class Customer, sau đây là nội dung file configuration metadata.

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

[*http://www.springframework.org/schema/context/spring-context-*](http://www.springframework.org/schema/context/spring-context-)

*4.3.xsd"*>

<context:annotation-config/>

<context:component-scan base-package=*"stu.ytbhoanghai.abc"*/>

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/></bean>

</beans>

Để ý chúng ta sẽ thấy, không tồn tại bean có id là customer trong file configuration. Và để sử dụng được các annotation trên, chúng ta phải thông báo thẻ <context:component-scan base-package="stu.ytbhoanghai.abc"/>. Spring scan qua các class trong package nếu tồn tại 1 trong 4 annotation trên, chúng sẽ được tự động tạo ra.

Sau đây là nội dung phương thức main để lấy ra bean có id là customer, thiết lập giá trị và in ra màn hình kết quả.

**private** **static** Customer *customer*;

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

ApplicationContext context = **new**

ClassPathXmlApplicationContext("spring-context.xml");

*customer* = (Customer) context.getBean("customer");

*customer*.setName("Hoang Hai");

System.***out***.println(*customer*);

((ClassPathXmlApplicationContext) context).close();

}

Kết quả

Customer [id=0, name=Hoang Hai, oder=Oder [data=This is a Data]]

##### @Qualifier

Để chỉ định chính xác bean nào sẽ được tiêm vào thuộc tính của lớp, ta đặt annotation @Qualifier phía trước thuộc tính được đặt annotation @Autowired.

@Component

**public** **class** Customer {

**private** **int** id;

**private** String name;

@Autowired(required = **false**)

@Qualifier("oder1") **private** Oder oder;

**public** Customer() {

**super**();

**this**.name = "Hoang Hai";

}

**public** Customer(Oder oder) {

**super**();

**this**.oder = oder;

}

//get and set

}

oder1 là id của bean sẽ được tiêm (inject) vào thuộc tính oder của bean customer nếu như trong file configuration metadata tồn tại nhiều hơn 1 bean là các sự thể hiện khác nhau của cùng một class, dưới đây là các bean được khai báo trong configuration.

<bean id=*"oder2"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data"*

type=*"java.lang.String"*/></bean>

<bean id=*"oder1"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"This is a Data 1"*

type=*"java.lang.String"*/></bean>

#### Thao tác với Properties File

Spring cung cấp cho chúng ta rất nhiều cách để thao tác với properties file. Sau đây 2 cách thường được sử dụng phổ biến.

##### PropertyPlaceholderConfigurer

Đây là cách đơn giản và được sử dụng thường xuyên nhất, đó chính là sử dụng đối tượng PropertyPlaceholderConfigurer.

Đầu tiên là tạo 1 file properties và lưu vào thư mục resources của project, đường dẫn chính xác là src/main/resources, string value đại diện cho đường dẫn đó là classpath:

Ví dụ, ta tạo 1 file có tên là info.properties và lưu vào thư mục trên. Nội dung như sau:

customer.id=18

customer.name=Nguyen Hoang Hai

oder.data=100 Cay Keo

Để đọc được nội dung file trên, ta tạo đối tượng PropertyPlaceholderConfigurer như là 1 bean trong file configuration metadata như sau:

<bean class=*"org.springframework.beans.factory.*

*config.PropertyPlaceholderConfigurer"*>

<property name=*"location"* value=*"classpath:info.properties"*/>

</bean>

##### Context Namespace

Với cách này, chúng ta phải khai báo namespace context trong thẻ bean với những dòng sau:

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.3.xsd"

Sau đây là nội dung file configuration metadata khi sử dụng context namespace để đọc file properties:

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

[*http://www.springframework.org/schema/context/spring-context-*](http://www.springframework.org/schema/context/spring-context-)

*4.3.xsd"*>

<context:property-placeholder location=*"classpath:info.properties"*/>

</beans>

Vẫn còn một số cách như sử dụng Util namespace hay annotation @PropertySource, nhưng chúng không được sử dụng phổ biến và trong phạm vi tài liệu này chỉ đề cập đến các cách thường được sử dụng, nên những cách vừa nêu trên (Util namespace, @PropertySource) sẽ không được đề cập.

##### Sử dụng properties trong file configuration metadata

Cú pháp để đọc giá trị trong file properties sẽ là ${property\_name}. Ví dụ, để đọc giá trị của key oder.data trong file info.properties ta làm như sau:

<bean id=*"oder"* class=*"stu.ytbhoanghai.abc.Oder"*>

<constructor-arg name=*"data"* value=*"${oder.data}"*

type=*"java.lang.String"*/>

</bean>

##### Sử dụng properties với annotation @Value

Với tập tin configuration metadata việc đọc giá trị trong file properties đã được trình bày ở trên, tại đây chúng ta sẽ sử dụng annotation @Value để đọc giá trị của file properties trong code java.

Chúng ta chỉ việc thêm annotation @Value trước thuộc tính chúng ta chỉ định sẽ lấy giá trị từ file properties, sau đó viết cú pháp thích hợp. Xem ví dụ dưới đây để hiểu rõ hơn:

@Value(value = "${customer.name}") **private** String name;

### Spring MVC

#### Tổng quan về Spring MVC

Spring MVC là một module con trong Spring MVC framework, cung cấp kiến trúc Model-View-Controller và các components sẵn có để sử dụng và phát triển ứng dụng web một cách linh hoạt.

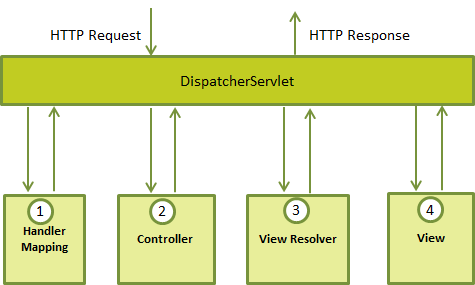
Mô hình MVC là kết quả của việc tách các khía cạnh khác nhau của ứng dụng (logic đầu vào, các xử lý logic, UI) trong khi đó cung cấp một sự kết hợp giữa các thành phần đó một cách “bớt chặt chẽ hơn”.

Model: Đóng gói dữ liệu ứng dụng và bao gồm các POJO.

View: Chịu trách nhiệm nhận các giá trị của model và vẽ ra trang HTML mà trình duyệt có thể hiển thị được.

Controller: Chịu trách nhiệm nhận và xử lý các yêu cầu từ người dùng và tạo các model phù hợp và trả về view.

Spring MVC được thiết kế xung quanh dispatcherServlet để xử lý tất cả các HTTP request và HTTP response. Luồng xử lý các yêu cầu của DispatcherServlet được minh họa theo hình:



Hình 3‑12 Luồng xử lý DispatcherServlet

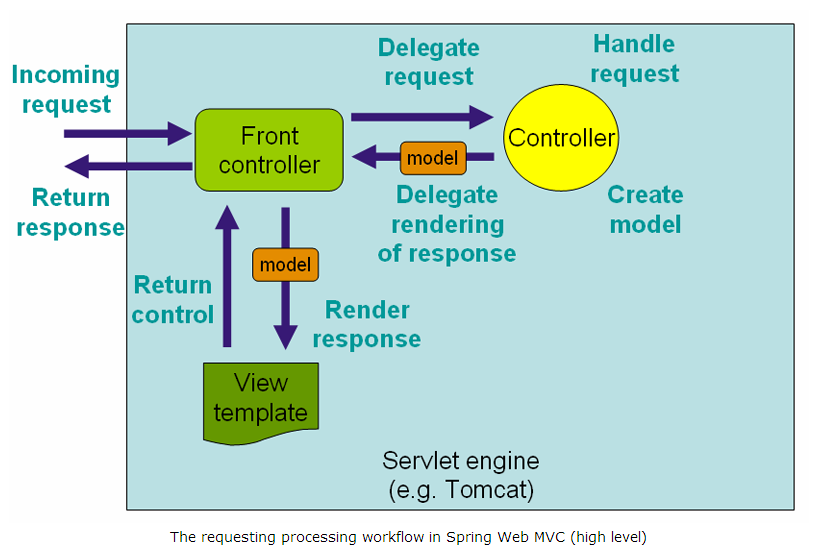
Sau đây sẽ mô tả chuỗi tác vụ tương ứng sẽ xảy ra khi Server nhận được một yêu cầu từ HTTP gửi đến dispatcherServlet:

* Sau khi nhận một HTTP request, DispatcherServlet gửi yêu cầu đến HandlerMapping (Thành phần này sẽ được trình bày trong mục [3.3.5.3]) để xác định controller nào sẽ xử lý yêu cầu này.
* Controller sẽ nhận các request và gọi các phương thức dịch vụ (service methods) phù hợp dựa trên việc sử dụng các method GET/POST… nếu yêu cầu đó cần truy xuất cơ sở dữ liệu thì Controller sẽ ủy nhiệm cho một business logic hay nhiều hơn một model để lấy thông tin và gửi dữ liệu về cho Controller. Lúc này Controller đóng gói mô hình dữ liệu và tên của một view sẽ được tải lên thành đối tượng ModelAndView và gửi trả về cho DispatcherServlet.
* DispatcherServlet gửi gói ModelAndView cho ViewResolver (Thành phần này sẽ được trình bày trong phần [3.3.5.5]) để tìm xem view (thông thường là một trang JSP, HTML) nào sẽ được load lên.
* Sau khi tất cả hoàn tất, DispatcherServlet sẽ chuyển dữ liệu từ model đến view và cuối cùng hiển thị lên trình duyệt.

#### DispathcherServlet

Spring MVC framework giống như nhiều web MVC frameworks, được thiết kế xung quanh một servlet trung tâm, nó gửi các request đến các controller và đưa ra những chức năng cho sự phát triển của ứng dụng web. Spring DispatcherServlet còn làm được nhiều hơn thế. Nó hoàn toàn được tích hợp với Spring IoC container và như vậy cho phép ta sử dụng bất cứ đặc tính nào mà Spring có.

Luồng xử lí request của Spring MVC DispatcherServlet được minh họa trong hình bên dưới. Nếu ta có hiểu biết về mô hình sẽ nhận thấy rằng DispatcherServlet là một sự diễn tả của mẫu thiết kế “Front Controller” (đây là một mẫu mà Spring MVC chia sẽ với nhiều web framework khác).



Hình 3‑13 Luồng xử lý một request trong Spring MVC

Như đã trình bày ở phía trên, DispathcherServlet là một servlet (Nó thực hiện kế thừa lớp HttpServlet), và như vậy nó được khai báo trong web.xml của ứng dụng web. Những request mà ta muốn DispatcherServlet xử lí sẽ được ánh xạ, sử dụng một URLmapping trong cùng file web.xml. Đây là chuẩn cấu hình của J2EE servlet. Dưới đây là ví dụ cụ thể về khai báo và định nghĩa DispatcherServlet.

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

File web.xml của một ứng dụng web sẽ được đặt trong đường dẫn như sau: src/main/webapp/WEB-INF.

Như ta thấy tất cả các request bắt đầu bằng / sẽ được FrontController (DispatcherServlet) là springDispatcherServlet xử lý.

Khi DispatcherServlet được khởi tạo, Spring sẽ gọi file có tên [tên-servlet]-servlet.xml trong thư mục WEB-INF để cấu hình. Trong ví dụ trên, Spring sẽ gọi file springDispatcherServlet-servlet.xml

Hoặc chúng ta có thể khai báo đường dẫn file cấu hình cho DispatcherServlet bằng cách khai báo thêm tag init-param như sau.

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/spring/appServlet/servlet-context.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

Khi đó Spring sẽ gọi đến file servlet-context.xml trong thư mục src/main/webapp/WEB-INF/spring/appServlet để cấu hình.

Từ đây trở đi, các đường dẫn có dạng src/main/webapp/ sẽ được thay thế bằng / để tránh nhẫm lẫn cũng như dễ dàng hơn trong cách trình bày.

#### Handler Mapping

Sử dụng một handler mapping chúng ta có thể ánh xạ một web request đến một handler thích hợp. Có một vài handler mapping chúng ta có thể sử dụng một cách tự nhiên, ví dụ: SimpleUrlHandlerMapping hoặc BeanNameUrlHandlerMapping.

Chức năng của một HandlerMapping cơ bản cung cấp là sự phân phối của một HandlerExecutionChain. Nó cần chứa handler phù hợp với request, và cũng có thể chứa đựng một danh sách các “handler chặn” mà được áp dụng cho request. “handler chặn” ở Servlet ta biết nó dưới cái tên Filter và trong Spring MVC nó có tên là interceptor.

Khi một request vào, DispatcherServlet sẽ điều khiển nó thông qua handler mapping để phân tích và kiểm tra request. Sau đó DispatcherServlet sẽ thực thi handler và các bộ chặn trong chuỗi thực thi nếu có.

Phần này mô tả hai handler mapping thông dụng. Chúng mở rộng từ AbstractHandlerMapping.

##### BeanNameUrlHandlerMapping

Thực hiện này của xử lý ánh xạ (mapping) phù hợp với URL của request với tên của controller beans. Các bean phù hợp sau đó được sử dụng như là bộ điều khiển cho các yêu cầu.

Chúng ta giả sử chúng ta có 3 trang web trong ứng dụng. URL tương ứng là:

* <http://localhost:8080/eShop/home.html>
* <http://localhost:8080/eShop/cart.html>
* <http://localhost:8080/eShop/shop.html>

Các Controller thực hiện các yêu cầu cho các trang web trên là:

* stu.ytbhoanghai.abc.controller.Home
* stu.ytbhoanghai.abc.controller.Cart
* stu.ytbhoanghai.abc.controller.Shop

Vì vậy chúng ta cần phải xác định các controller điều khiển trong tập tin ngữ cảnh ứng dụng (spring application context). Tức là tên của bean controller phù hợp với request. Các controller bean đó trong tập tin cấu hình xml sẽ trông như dưới đây.

<bean id=*"handlerMapping"* class=

*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*/>

<bean name=*"/home.html"* class=*"stu.ytbhoanghai.abc.controller.Home"* />

<bean name=*"/cart.html"* class=*"stu.ytbhoanghai.abc.controller.Cart"* />

<bean name=*"/shop.html"* class=*"stu.ytbhoanghai.abc.controller.Shop"* />

Khi đó, khi có các request dạng /home.html /cart.html hoặc /shop.html sẽ được xử lý bởi các Controller tương ứng là Home, Cart và Shop. Chúng ta để ý rằng tên của các bean sẽ phải phù hợp với URL request.

##### SimpleUrlHandlerMapping

BeanNameUrlHandlerMapping ở trên đặt một sự hạn chế về tên của các controller bean sao cho phù hợp với URL của request. SimpleUrlHandlerMapping đã loại bỏ những hạn chế này và sử dụng cơ chế mapping để ánh xạ các controller bean với URL sử dụng thuộc tính mappings.

<bean class=

*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/welcome"*>Welcome</prop>

<prop key=*"/\*/cart"*>GioHang</prop>

<prop key=*"/shop"*>CuaHang</prop>

</props>

</property>

</bean>

<bean name=*"Welcome"* class=*"stu.ytbhoanghai.abc.controller.Home"* />

<bean name=*"GioHang"* class=*"stu.ytbhoanghai.abc.controller.Cart"* />

<bean name=*"CuaHang"* class=*"stu.ytbhoanghai.abc.controller.Shop"* />

#### Controller

Controller là các thành phần được DispatcherServlet gọi để xử lý các nghiệp vụ business. Tất cả các Controller trong Spring đều implement từ Controller interface trong package org.springframework.web.servlet.mvc

Controller interface định nghĩa ra 1 phương thức chịu trách nhiệm xử lí 1 request và trả về model và view thích hợp. Ba khái niệm này chính là nền tảng của Spring MVC framework: Model – View – Controller.

Kể từ khi Spring 2.5 giới thiệu và đưa vào mô hình lập trình cho các MVC controller dựa trên các annotation như @Controller, @RequestMapping, @RequestParam, @PathVariable, … thì việc tạo ra một controller trở nên vô cùng đơn giản.

Để tạo ra một controller chúng ta không cần kế thừa từ một lớp cơ sở hoặc thực thi từ bất kỳ interface nào. Hãy xem xét 2 ví dụ dưới đây.

* Tạo controller bằng cách thực thi giao diện AbstractController

**public** **class** HomeController **implements** AbstractController{

**public** ModelAndView handleRequestInternal(

HttpServletRequest request,

HttpServletResponse response) **throws** Exception {

**return** **new** ModelAndView("home");

}

}

Khi sử dụng AbstractController làm lớp cơ sở cho các controller, chỉ cần ghi đè phương thức handleRequestInternal(HttpServletRequest, HttpServletResponse), viết code xử lý, và trả về một đối tượng ModelAndView.

* Tạo controller bằng annotation

Vì Spring hỗ trợ mạnh mẽ cho các annotation nên đây là cách khuyến khích được sử dụng trong Spring MVC, bản thân nhận thấy sự tiện ích mà annotation mang lại vì thế trong tài liệu này sẽ sử dụng annotation để định nghĩa các Controller.

Sau đây là một ví dụ về tạo Controller bằng annotation mà Spring cung cấp:

@Controller(value = "/")

**public** **class** BaseController {

@RequestMapping(value = {"home", "index"})

**public** String index() {

**return** "index";

}

}

Như chúng ta đã thấy, với 2 annotation là @Controller và @RequestMapping là chúng ta đã có thể tạo ra một Controller một cách dễ dàng. Ngoài ra, chúng ta cũng không cần phải khai báo handler mapping như trong ví dụ 1, các request được mapping với controller và các phương thức của chúng qua giá trị của thuộc tính value được cung cấp. /home hay /index sẽ được xử lý bởi BaseController.

#### ViewResolver

Phần [3.3.5.1] (Tổng quan về Spring MVC) đã đề cập đến ViewResolver. Nhiệm vụ của nó là phân giải các view trong Spring được gán “View\_Name” để xác định view nào sẽ được load lên.

Sau đây là các ViewResolver trong Spring:

* BeanNameViewResolver:
* FreeMarkerViewResolver
* InternalResourceViewResolver
* JasperReportsViewResolver
* ResourceBundleViewResolver
* UrlBasedViewResolver
* VelocityLayoutViewResolver
* VelocityViewResolver
* XmlViewResolver
* XsltViewResolver

Ở đây, xin trình bày InternalResourceViewResolver vì nó được sử dụng nhiều nhất. Ngoài ra, nếu cá nhân đọc có nhu cầu tìm hiểu về các ViewResolver khác có thể tìm hiểu tại địa chỉ sau: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ViewResolver.html.

InternalResourceViewResolver sẽ ánh xạ Logical name của Object ModelAndView được trả về bởi Controller đến Physical View.

<bean class=

*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/views/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

Như ví dụ trên, nói một cách ngắn gọn. Các Physical View được đặt trong /WEB-INF/views có kiểu là jsp sẽ được gọi mỗi khi Controller trả về Logical name phù hợp với nó.

@RequestMapping(value = "/cart")

**public** String cart(Model model) {

**return** "cart";

}

Nếu chúng ta trả về 1 String mà không phải là một đối tượng ModelAndView thì cũng sẽ không có vấn đề gì xảy ra vì trả về ModelAndView là cách mà chúng ta thường dùng khi làm việc với Spring 2.0, ở Spring 3.0 chuỗi được trả về sẽ quy định logical view name, và để lấy đối tượng model ta chỉ việc thêm tham số Model như trên ví dụ. Thật ra thì không có quy định cụ thể nào, đây chỉ là thông tin thêm. Chúng ta có thể sử dụng bất cứ cách nào mà chúng ta thấy thoải mái.

@RequestMapping(value = "/cart")

**public** ModelAndView cart() {

ModelAndView modelAndView = **new** ModelAndView();

modelAndView.setViewName("cart");

**return** modelAndView;

}

Và khi đó, view có tên là cart.jsp trong thư mục /WEB-INF/views sẽ được DispathcherServlet gọi để truyền Model vào và gửi về trình duyệt.

#### Tìm hiểu về các annotation trong Spring MVC

Phần bên trên là lý thuyết cơ bản mà chúng ta cần phải hiểu để làm việc được với Module Spring MVC của Spring Framework.

Sau đây sẽ giới thiệu các annotation mà chúng ta sẽ phải thường xuyên sử dụng khi làm việc với Spring MVC kèm theo ví dụ.

##### @RequestMapping

Đây là một trong những annotation được sử dụng nhiều nhất trong Spring MVC, nó có nhiệm vụ mapping các request với các lớp hoặc phương thức xử lý request đó.

@RequestMapping(value = "/cart")

**public** String cart() {

**return** "cart";

}

Đây là 1 phương thức sẽ xử lý các request tới URI /cart. Nó có nhiệm vụ trả về một Logical View Name là cart.

Nó cũng có thể xử lý nhiều URI như sau:

@RequestMapping(value = {"/", "/home", "/index"})

**public** String index() {

**return** "index";

}

##### @PathVariable

Annotation @PathVariable được sử dụng để xử lý những URI động, có một hoặc nhiều paramter bên trong URI.

@RequestMapping(value = "/shop/{name}/{id}")

**public** String products( @PathVariable String name,

@PathVariable **int** id) {

// Các xử lý sẽ thực hiện ở đây.

**return** "shop";

}

##### @RequestParam

Khi request có method là GET, các giá trị đi kèm sẽ được đặt phía sau cùng của URL. Chúng ta sử dụng annotation @RequestParam để bắt lấy những giá trị đó.

Ví dụ: URL có dạng 127.0.0.1:8080/eShop/shop?name=kit&id=10

Khi đó, các Param name và id sẽ được bắt lấy như sau:

@RequestMapping(value = "/shop", method = RequestMethod.***GET***)

**public** String products( @RequestParam String name,

@RequestParam **int** id) {

// Các xử lý sẽ được thực hiện ở đây.

**return** "shop";

}

##### @ResponeBody và @RestController

Khi chúng ta phát triển web service API các giá trị trả về không phải là view mà là các giá trị json, xml, text… Annotation @ResponeBody được thêm vào trước các phương thức của các Controller để chỉ dẫn ra rằng phương thức này sẽ trả về text thay vì là view.

@RequestMapping(value = "/shop")

@ResponseBody **public** String products() {

**return** "shop";

}

Như ví dụ ở bên trên, giá trị trả về là chuỗi shop.

@RestController đơn giản là @ResponeBody + @Controller, nó được đặt phía trước các lớp được định nghĩa là Controller để quy định các giá trị mà các phương thức bên trong nó trả về là text chứ không phải view.

// @Controller @ResponseBody => Đã được chú thích lại

@RestController

**public** **class** BaseController {…}

Chúng ta hoàn có thể chỉ định kiểu giá trị được trả về ngoài kiểu text, đối với xml và json bằng cách khai báo thêm các thư viện hỗ trợ chuyển đổi 1 object thành các file xml hoặc json tương ứng.

Đối với json là các thư viện Jackson Core, Jackson Databind, Jackson Annotations. Xml đã được hỗ trợ bởi thư viện spring-webmvc nên chúng ta không cần khai báo thêm mà chỉ sử dụng annotation đã được cung cấp sẵn. Vì webservice không nằm trong phạm vi đồ án này nên phần giới thiệu chỉ dừng lại ở đây. Chúng ta có thể gặp lại các annotation này trong 1 phần nhỏ của chương trình demo đồ án khi mà nó kết hợp với AJAX để thực hiện các thao tác không đồng bộ với lời gọi tới API của server để tải file xml và phân tích cũng hiển thị chúng mà không cần tải trang web.

##### @ModelAttribute

Đại diện cho 1 thuộc tính của đối tượng Model (Model trong ModelAndView).

Khi Form từ View thực hiện việc submit một đối tượng modelAttribute, ta sẽ “bắt” lấy đối tượng đó ở method của Controller bằng annotation @ModelAttribute: Sau đây là 1 ví dụ:

<%@taglib uri=*"http://www.springframework.org/tags/form"*

prefix=*"form"*%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Employee</h2>

<form:form method=*"POST"* action=*"addEmployee"*

modelAttribute=*"employee"*>

Id: <form:input path=*"id"* />

<br/>

Name: <form:input path=*"name"* />

<input type=*"submit"* value=*"Submit"* />

</form:form>

</body>

</html>

Khi đó, Phương thức “bắt” employee được “quăng” lên Controller sẽ có dạng:

@RequestMapping(value = "/addEmployee", method = RequestMethod.***POST***)

**public** String doAddEmployee(@ModelAttribute("employee") Employee employee) {

// Xử lý dữ liệu tại đây.

**return** "view-employee";

}

@Valid, @Validate

Spring hỗ trợ chúng ta validate các thuộc tính khi submit form bằng cách sử dụng các annotation validate của hibernate-validator, javax-validation.

Hai thư viện validation-api và hibernate-validator cung cấp các annotation định nghĩa cho việc validate. Chúng ta có thể thêm trực tiếp vào project bằng cách tải thư viện về, hoặc khai báo dependency của chúng như đã được trình bày ở phần [2.2.4].

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.0.18.Final</version>

</dependency>

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>2.0.1.Final</version>

</dependency>

Chúng ta có Model User được Validate như sau:

**public** **class** User {

@NotNull(message = "Name may not be Null")

**private** String name;

@NotBlank @Length(min = 5, max = 10)

**private** String password;

@NotEmpty @Pattern(regexp = "dd/MM/yyyy")

**private** Date date;

//get and set

}

Các thông điệp in ra thông báo lỗi sẽ được lấy ra từ 3 nguồn với mức ưu tiên như sau, trong file properties định nghĩa thông báo lỗi, trong thuộc tính message của annotation, và giá trị mặc định của hibernate validate.

Sau đây là ví dụ về 1 file properties định nghĩa thông báo lỗi. Để sử dụng được nó, trước hết chúng ta phải khai báo 1 bean chỉ định vị trí đặt file properties như sau:

<bean id=*"messageSource"* class=*"org.springframework*

*.context.support.ReloadableResourceBundleMessageSource"*>

<property name=*"basename"* value=*"/i18n/message"* />

</bean>

Nội dung file message.properties trong thư mục src/main/webapp/i18n như sau:

## Chỉ rõ thông báo lỗi cho 1 đối tượng có tên cụ thể “user”

NotNull.user.name=Name can not be Null

NotBlank.user.password=Password can not be Empty

NotEmpty.user.date=Date can not be Empty

## Lỗi chung cho tất cả các đối tượng được đặt annotation tương ứng

## NotNull=Not Null

## NotBlank=Not Blank

## Lỗi cho 1 annotation tự tạo

## Phone=Phone number is not valid

Ngoài ra chúng ta cũng có thể tự định nghĩa các annotation validate.

Định nghĩa khai báo annotation Phone:

@Documented

@Constraint(validatedBy = PhoneValidator.**class**)

@Retention(RetentionPolicy.***RUNTIME***)

@Target({ ElementType.***FIELD***, ElementType.***METHOD*** })

**public** **@interface** Phone {

String message() **default** "{Phone}";

Class<?>[] groups() **default** {};

Class<? **extends** Payload>[] payload() **default** {};

}

Định nghĩa annotation Phone:

**public** **class** PhoneValidator **implements**

ConstraintValidator<Phone, String> {

**public** **void** initialize(Phone paramA) {}

**public** **boolean** isValid(String phoneNo

, ConstraintValidatorContext ctx) {

**if** (phoneNo == **null**) {

**return** **false**;

}

**return** phoneNo.matches("\\d{10}");

}

}

Áp dụng Validation:

@RequestMapping(value = "/login")

**public** String login(@ModelAttribute(value = "user") @Valid User user,

BindingResult bindingResult) {

**if**(bindingResult.hashErrors()) {

**return** "login";

}

**return** "profile";

}

Lưu ý rằng annotation @Valid hoặc @Validate đều được nhưng nó phải nằm trước đối tượng được Validate, các thông báo lỗi từ Validate sẽ được đại diện bởi đối tượng BindingResult. Đây là một trong những trường hợp hiếm về thứ tự của các tham số trong Spring MVC.

##### @ExceptionHandler

@ExceptionHandler(NullPointerException.**class**)

**public** String processNullPointerException() {

**return** "null-pointer-exception";

}

Annotation @ExceptionHandler dùng để chỉ dẫn rằng phương thức processNullPointerException sẽ xử lý cho các Exception NullPointerException được ném ra từ các phương thức được đánh dấu @RequestMapping trong cùng một Class.

##### @ControllerAdvice

Khi sử dụng annotation @ControllerAdvice cho class, thì các method được đánh dấu @ExceptionHandler trong class đó có thể xử lý exception cho cả các method của các class khác hoặc package khác.

@ControllerAdvice

**public** **class** GlobalExceptionHandling {

//...

}

Ví dụ trên là xử lý cho tất cả các class. Ví dụ dưới đây là xử lý cho 1 package cụ thể:

@ControllerAdvice(basePackages="stu.ytbhoanghai.abc.controller")

**public** **class** GlobalExceptionHandling {

//...

}

### Spring Hibernate

Spring cung cấp các API giúp lập trình viên dễ dàng tích hợp Spring với ORM Framework như JPA (Java persistence API), JDO (Java data object), Oracle Toplink, IBatis… Hibernate là 1 framework thực thi JPA nó bao gồm những đặc tả ORM cốt lõi của bản thân hibernate và thực thi lại giao diện JPA.

Ở đây sẽ không nói quá nhiều lý thuyết về hibernate mà mặc định cá nhân người đọc đã có kiến thức nền tảng cơ bản về hibernate. Phần này sẽ trình bày mã và giải thích về những ưu điểm khi kết hợp 2 Framework trên.

Chúng ta cũng biết trong Hibernate đối tượng thực hiện interface SessionFactory sẽ là đối tượng nắm giữ các thông tin liên quan đến kết nối cũng như thông tin về các entity để thao tác với database. Vì nó là một đối tượng nặng (heavy weight object) và là một luồng an toàn (Thread-safe) nên sẽ tốt hơn nếu đối tượng này được quản lý bởi Spring. Phần này sẽ trình bày cách mà chúng ta kết hợp Spring Framework với Hibernate Framework và những ưu điểm mà nó mang lại so với khi không dùng Spring.

#### Các thư viện.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.2.1.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>5.2.1.RELEASE</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.4.8.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.18</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

Phần phía trên cung cấp các thư viện mà chúng ta dùng để kết hợp Spring Framework với Hibernate. Trong đó spring-context là thư viện của spring, hibernate-core là bộ đặc tả ORM cố lõi của hibernate, spring-orm cung cấp các API giúp tích hợp Spring với ORM Framework mà cụ thể ở đây là hibernate. Một số thư viện database connection pooling được hỗ trợ bởi hibernate như C3P0, Apache DBCP, Proxool… ở đây chúng ta sẽ dùng apache dbcp. Tất nhiên là không thể thiếu jdbc cho mysql.

#### Cấu hình

Như đã nói ở trên (phần giới thiệu [3.3.6]) Khi làm việc với JPA chúng ta có file cấu hình thường được đặt tên là persistence.xml tương tự như thế với Hibernate là hibernate.cfg.xml. Đối tượng SessionFactory hay EntityManagerFactory sẽ đọc các thông tin cấu hình này, thực hiện tạo các kết nối đến database và công đoạn này tốn rất nhiều thời gian. Chúng ta thường tạo ra các lớp util cung cấp các đối tượng trên theo thiết kế singleton pattern như sau:

**public** **class** HibernateUtil {

**private** **static** SessionFactory *sessionFactory*;

**static** {

*sessionFactory* = **new** Configuration()

.configure()

.buildSessionFactory();

}

**public** **static** SessionFactory getSessionFactory() {

**return** *sessionFactory*;

}

}

Đối với Spring, chúng ta không cần tạo lớp util đó, chỉ cần khai báo sessionFactory trong spring context và gọi ra khi cần thiết. Điều này nhắc nhớ chúng ta về cách mà Spring quản lý scope của các bean đã được trình bày ở mục [3.3.4.1] (Scope của bean), và vì là không định nghĩa gì về scope của bean nên scope mặc định của bean sessionFactory sẽ là Singleton như bên dưới đây.

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${driverClassName}"*/>

<property name=*"url"* value=*"${url}"*/>

<property name=*"username"* value=*"${username}"*/>

<property name=*"password"* value=*"${password}"*/>

</bean>

<bean id=*"sessionFactory"*

class=*"org.springframework.orm.hibernate5.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*/>

<property name=*"packagesToScan"*

value=*"stu.ytbhoanghai.abc.entity"*/>

<property name=*"hibernateProperties"*

value=*"classpath:hibernate.properties"*/>

</bean>

Phần mã phía trên thay thế cho file hibernate.cfg.xml. Như đã đề cập ở mục [3.3.4.3] (Dependency Injection) chúng ta có nhiều cách để lấy 1 bean ra khỏi spring context như là dùng phương thức getBean như sau:

**private** **static** ClassPathXmlApplicationContext *context*;

**static** {

*context* = **new** ClassPathXmlApplicationContext("application-

context.xml");

}

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

SessionFactory sessionFactory = (SessionFactory)

*context*.getBean("*sessionFactory*");

}

Hoặc đơn giản là dùng annotation @Autowired

@Repository

@Transactional(rollbackFor = Exception.**class**)

**public** **abstract** **class** DAO<T **extends** Serializable> {

@Autowired

**protected** SessionFactory sessionFactory;

**public** **void** insert(T object) {

Session session = sessionFactory.getCurrentSession();

session.save(object);

}

}

#### Transaction

Chúng ta cũng biết trong hibernate các phương thức mà Interface Session cung cấp như save, saveOrUpdate, update, merge, persist, delete… Không thực sự lưu giữ đối tượng vào database mà chỉ chuyển trạng thái của chúng qua lại giữa transient, persistent và detached các đối tượng (Object) được quản lý bởi Session sẽ được lưu trong bộ nhớ cache cấp 1 của hibernate vùng đó được gọi là Persistentce Context. Các đối tượng trong Persistentce Context chỉ thực sự được lưu vào database khi ta thực hiện commit 1 transation.

Session sess = factory.openSession();

Transaction tx;

**try** {

tx = sess.beginTransaction();

//do some work

tx.commit();

}

**catch** (Exception e) {

**if** (tx!=**null**) tx.rollback();

**throw** e;

}

**finally** {

sess.close();

}

Đoạn mã trên là cần thiết nhưng các thao tác luôn phải lặp đi lặp lại giữa các phương thức và chỉ khác nhau ở phần //do some work.

Mở 1 Session mà không đóng thì sẽ, hoặc quăng ra một ngoại lệ hoặc có rất nhiều thao tác không mong muốn vô tình được thực hiện trên session đó. Đôi khi 1 Transaction quên commit thì rất khó để dò lỗi vì không có exception bị ném ra.

Khai báo 1 đối tượng HibernateTransactionManager trong spring-context và đặt một tag <tx:annotation-driven /> có nội dung như sau:

<tx:annotation-driven transaction-manager=*"transactionManager"* />

<bean id=*"transactionManager"*

class=*"org.springframework.orm.hibernate5.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"* />

</bean>

Sau đó, tại lớp hoặc tại phương thức mà chúng ta muốn thực hiện transaction, đặt annotation @Transactional(rollbackFor = Exception.class) thuộc tính rollbackFor là không bắt buộc, nó giúp quy định khi xảy ra ngoại lệ nào thì sẽ rollback lại transaction.

@Repository

@Transactional(rollbackFor = Exception.**class**)

**public** **abstract** **class** DAO<T **extends** Serializable> {

@Autowired

**protected** SessionFactory sessionFactory;

**public** **void** insert(T object) {

Session session = sessionFactory.getCurrentSession();

session.save(object);

}

}

Chúng ta không phải thao tác (getTransaction, beginTransaction, commitTransaction, hay rollback nữa) thao tác đã được lược bỏ đi rất nhiều. Và vì các phương thức trong 1 transaction nên ta dùng getCurrentSession và vì thế cũng không cần phải thực hiện close 1 session.

Đó là những lợi ích cơ bản nhất mà chúng ta có thể dễ dàng thấy khi kết hợp Spring Framework với hibernate. Mỗi người sẽ có 1 cách nhìn nhận khác nhau về lợi ích mà chúng mang lại khi thao tác. Kết thúc phần này cũng đã khép lại chương 2 của đồ án.