

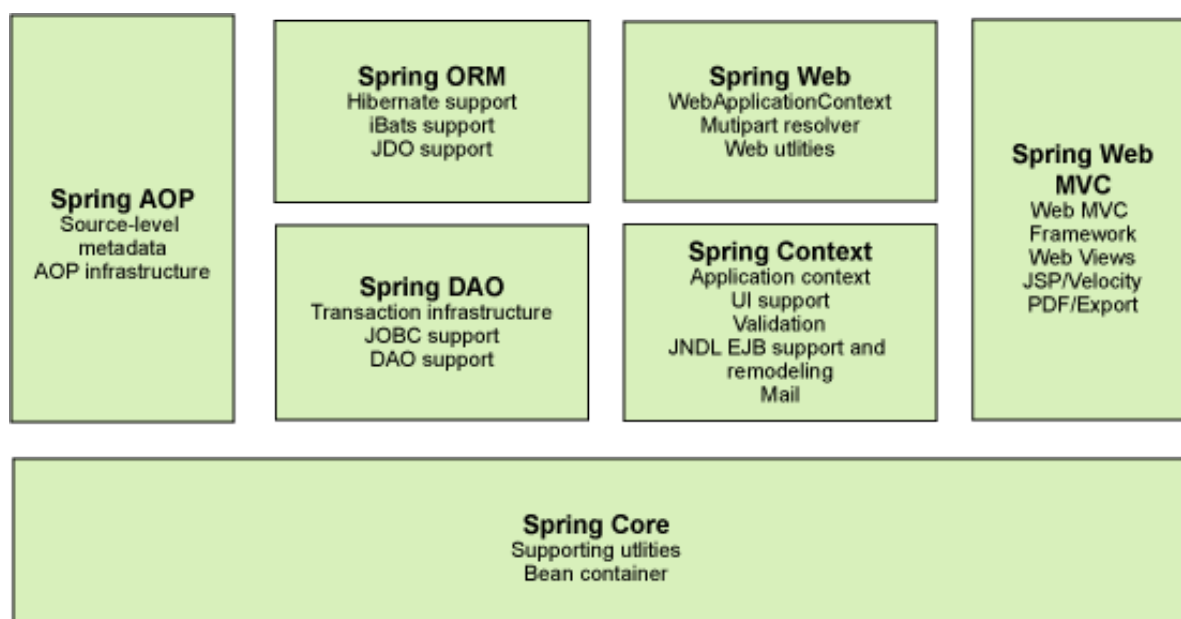
CHƯƠNG 5: SPRING MVC

5.1 Giới thiệu Spring

Spring Framework là một framework mã nguồn mở được viết bằng Java. Nó được xem như là một giải pháp kiến trúc tốt nhất của Java EE hiện nay.

Theo thiết kế, framework này giảm nhẹ công việc kỹ thuật cho lập trình viên Java, để họ tập trung sâu vào các công việc nghiệp vụ của ứng dụng. Đồng thời cung cấp một giải pháp toàn diện để họ thực hiện ứng dụng một cách tiện lợi nhất, chặt chẽ nhất đồng thời dễ dàng bảo trì, bảo dưỡng sau này.

Các module chính: Spring được tổ chức thành 7 modules :



5.1.1 Mô hình MVC

❖ Spring Core

Core package là phần lõi của framework, cung cấp những đặc tính IoC (Inversion of Control) và DI (Dependency Injection).

BeanFactory đảm nhận việc sản sinh và móc nối sự phụ thuộc giữa các đối tượng trong file cấu hình.

❖ Spring Context/Application Context

Phía trên của Core package là Context package - cung cấp cách để truy cập đối tượng.

Context package kế thừa các đặc tính từ bean package và thêm vào chức năng đa ngôn ngữ (I18N), truyền sự kiện, resource-loading,...

❖ Spring AOP (Aspect Oriented Programming).

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 (chú thích) vào source code để hướng dẫn Spring và làm thế nào để thực hiện các phương thức sự kiện đã được cài đặt sẵn.

❖ **Spring DAO (Data Access Object)**

DAO package cung cấp cho tầng JDBC, bỏ bớt những coding dài dòng của JDBC và chuyển đổi mã lỗi được xác định bởi database vendor. JDBC package cung cấp cách lập trình tốt như declarative transaction management.

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 (Object Relational Mapping)**

ORM package cung cấp tầng tích hợp với object-relational mapping API bao gồm: JDO, Hibernate, iBatis.

Sử dụng ORM package bạn có thể sử dụng tất cả các object-relational mapping đó kết hợp với tất cả các đặc tính của Spring như declarative transaction management.

❖ **Spring Web module.**

Spring Web package cung cấp đặc tính của web như: chức năng file-upload, khởi tạo IoC container sử dụng trình lắng nghe servlet và web-oriented application context.

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.**

Spring Framework là một ứng dụng mã nguồn mở phổ biến để phát triển ứng dụng Java EE dễ dàng hơn. Nó là một container gồm Web Framework (tiếp nhận và xử lý yêu cầu, chia sẻ dữ liệu...) và Web View (quản lý giao diện).

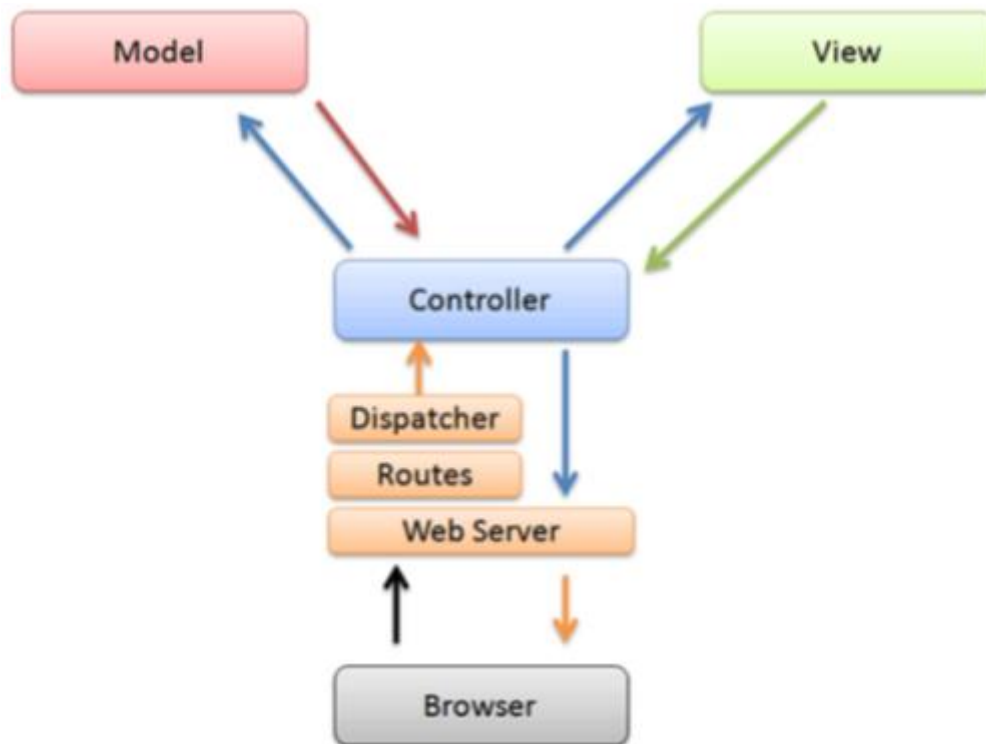
MVC Framework được cài đặt đầy đủ các đặc tính của MVC pattern để xây dựng các ứng dụng Web. Các thành phần gồm View (JSP, Velocity, Tiles và generation of PDF và Excel file.), Model (domain model) và Controller (chứa các xử lý yêu cầu).

5.1.2 Lợi ích của Spring MVC

- ✓ Tất cả các framework đã được tích hợp rất tốt vào Spring.
- ✓ Hoạt động rất tốt khi áp dụng theo kiến trúc MVC.
- ✓ Sử dụng cơ chế plug-in.

- ✓ Kết hợp rất tốt với các O/R (object-relational) Mapping frameworks như là Hibernate.
- ✓ Dễ Testing ứng dụng.
- ✓ Ít phức tạp hơn so với các framework khác.
- ✓ Cộng đồng người sử dụng rất nhiều, nhiều sách mới được xuất bản.

5.1.3 Mô hình hoạt động Spring MVC



Hình: Mô hình MVC

❖ Model

Model gồm các lớp java có nhiệm vụ:

- ✓ Biểu diễn data và cho phép đọc/ghi data thông qua các phương thức getter/setter theo qui ước trong JavaBean.
- ✓ Buộc dữ liệu form giao diện tức là nhận dữ liệu từ tham số và cung cấp dữ liệu để trình bày lên giao diện.
- ✓ Thi hành các yêu cầu (tính toán, kết nối CSDL ...)
- ✓ Trả về các giá trị tính toán theo yêu cầu của Controller

❖ View

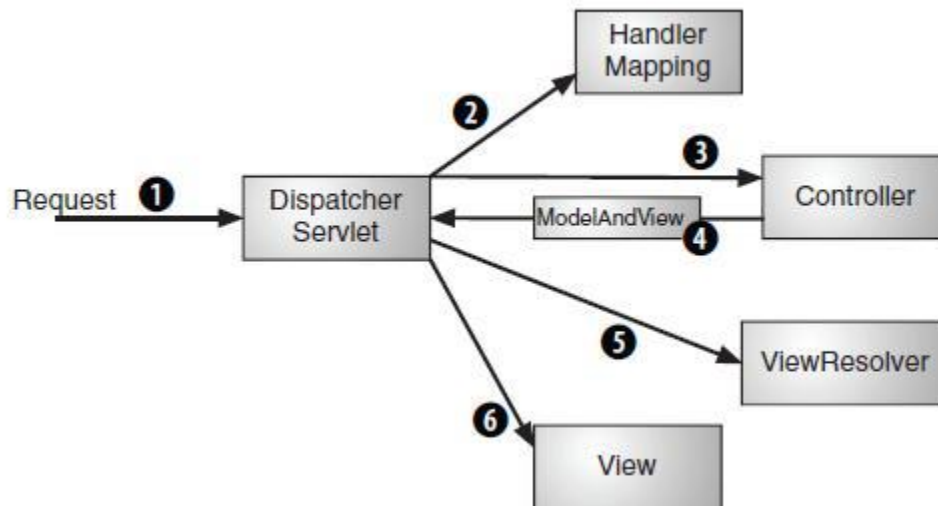
Bao gồm các mã tương tự như JSP, HTML, CSS, XML, Javascript, JSON... để hiển thị giao diện người dùng, các dữ liệu trả về từ Model thông qua Controller...

❖ Controller

Đồng bộ hoá giữa View và Model. Tức là với một trang JSP này thì sẽ tương ứng với lớp java nào để xử lý nó và ngược lại, kết quả sẽ trả về trang jsp nào. Nó đóng vai trò điều tiết giữa View và Model.

Như vậy, chúng ta có thể tách biệt được các mã java ra khỏi mã html. Do vậy, nó đã giải quyết được các khó khăn sự phụ thuộc nghiệp vụ lẫn nhau giữa các thành viên tham gia dự án. Người thiết kế giao diện và người lập trình java có thể mang tính chất độc lập tương đối. Việc kiểm lỗi hay bảo trì sẽ dễ dàng hơn, việc thay đổi các theme của trang web cũng dễ dàng hơn ...

5.1.4 Quy trình xử lý request



Cũng giống như các java-base MVC Framework khác Spring MVC cũng phải requests thông qua một front controller servlet. Một bộ front controller servlet đại diện duy nhất chịu trách nhiệm về yêu cầu các thành phần khác của một ứng dụng để thực hiện việc xử lý thực tế. Trong trường hợp của Spring MVC, DispatcherServlet là bộ điều khiển phía trước.

Một Request được gửi bởi DispatcherServlet đến điều khiển (được chọn thông qua một bản đồ xử lý). Một khi điều khiển kết thúc, yêu cầu sau đó được gửi để xem (đó là lựa chọn thông qua ViewResolver) để làm cho đầu ra.

Trình tự trình xử lý yêu cầu:

1. DispatcherServlet tiếp nhận Request từ trình duyệt
2. DispatcherServlet gửi yêu cầu đến Handler Mapping (bộ phận quản lý các ánh xạ) để xác định controller nào sẽ xử lý yêu cầu này.
3. DispatcherServlet gửi yêu cầu đến Controller sau khi biết được Controller nào sẽ xử lý yêu cầu. Nếu yêu cầu đó cần truy xuất cơ sở dữ liệu thì Controller sẽ ủy nhiệm cho một business logic hay nhiều hơn một service Objects (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.
4. Gói ModelAndView được gửi trả về DispatcherServlet.
5. DispatcherServlet gửi gói ModelAndView cho ViewResolver để tìm xem trang web (JSP) nào sẽ được load lên.
6. DispatcherServlet load trang web đó lên cùng với dữ liệu của nó và trả kết quả về cho trình duyệt.

5.1.5 Cấu hình DispatcherServlet

DispatcherServlet đóng vai trò như trái tim của Spring MVC. Servlet này có chức năng là front controller. Giống như bất kỳ servlet khác, DispatcherServlet phải được cấu hình trong web.xml file.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Theo mặc định, khi DispatcherServlet được nạp, nó sẽ tải các ứng dụng Spring context từ file XML có tên dựa vào tên của servlet.

Trong trường hợp này, vì servlet có tên dispatcher vì vậy DispatcherServlet sẽ cố gắng tải các ứng dụng context từ một file có tên dispatcher-servlet.xml.

Sau đó, bạn phải chỉ ra mẫu URL sẽ được xử lý bởi DispatcherServlet. Hãy thêm <servlet-mapping> sau đây vào web.xml để cho DispatcherServlet xử lý tất cả các URL kết thúc bằng ".htm":

```
<servlet-mapping>
    <servlet-name> dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

5.1.6 Cấu hình trên nhiều file

Trong thực tế bạn nên chia cấu hình ứng dụng của bạn thành nhiều file, mỗi file cấu hình một vấn đề khác nhau để dễ dàng quản lý

Tập tin cấu hình	Nội dung cấu hình
Spring-config-mvc.xml	Cấu hình thông thường Spring MVC
Spring-config-mail.xml	Cấu hình bean cho phép gửi mail
Spring-config-upload.xml	Cấu hình bean cho phép upload file

Việc còn lại là làm thế nào để nạp tất cả các file cấu hình đó vào ứng dụng web

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-config-*.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
```

```
<servlet-name>spring</servlet-name>
<url-pattern>*.htm</url-pattern>

</servlet-mapping>
```

5.1.7 Xây dựng một Controller đơn giản

MyController lấy một thông báo Welcome to Spring MVC để hiển thị trên View

```
package nhatnghe.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

    @RequestMapping(value="myAction")
    public String myAction(ModelMap model) {
        model.addAttribute("message", "Welcome to Spring MVC");
        return "MyView";
    }
}
```

MyController chứa một action có tên MyAction phục vụ yêu cầu khẩn cầu với hình thức GET được ánh xạ với phương thức xử lý action có tên myActionMethod.

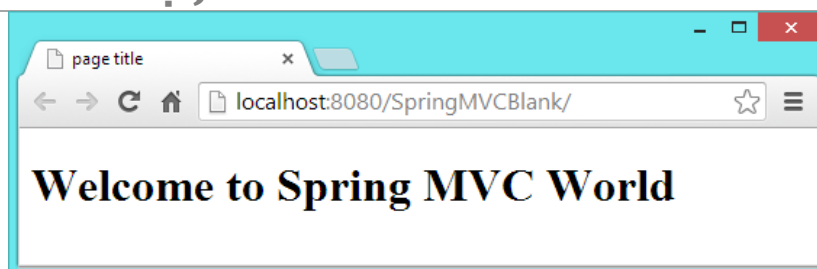
Phương thức này bổ sung một thuộc tính có tên message và giá trị là "Welcome to Spring MVC" vào model. Thuộc tính này sẽ được chia sẻ với View có tên MyView được trả về bởi phương thức.

Trên MyView có thể sử dụng biểu thức EL để truy xuất và hiển thị thuộc tính này: `${message}`

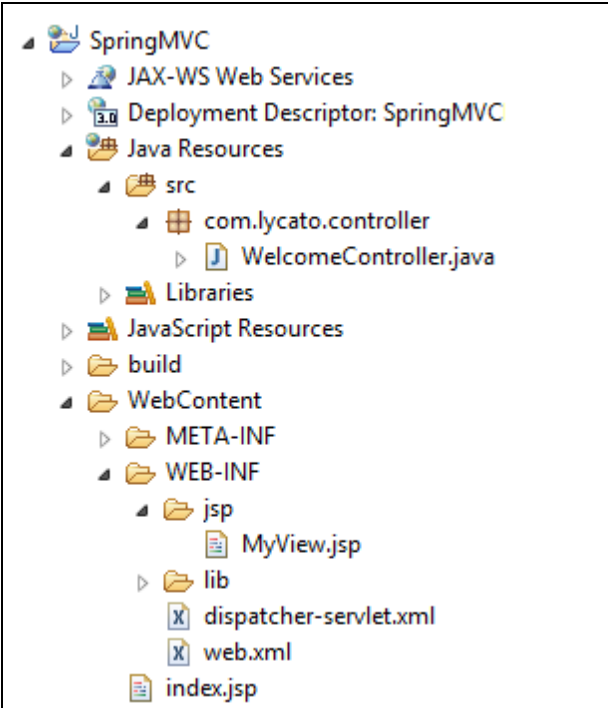
```
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC</title>
</head>
<body>
    <h2>${message}</h2>
</body>
</html>
```

5.1.8 Thực hành

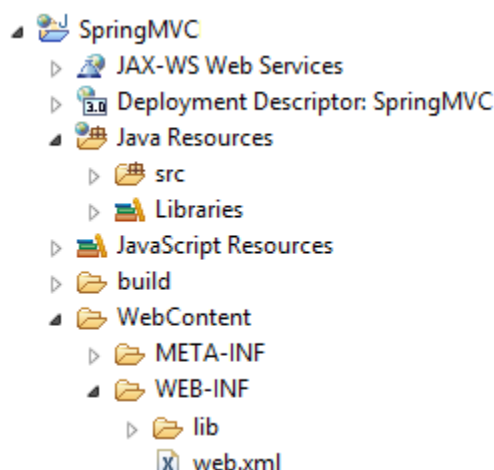
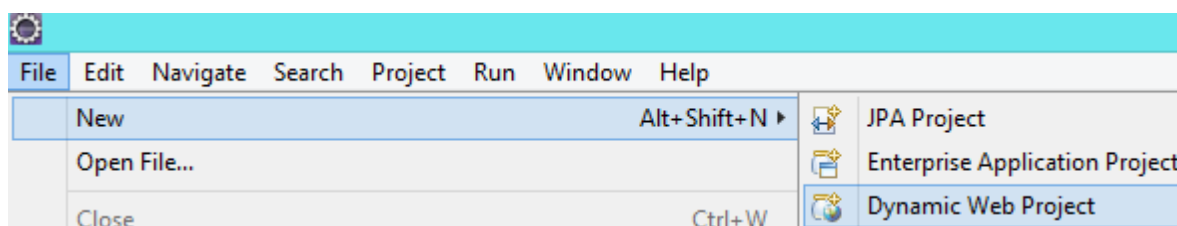
Để khám phá các thành phần của một dự án Spring MVC trong eclipse, chúng ta tạo dự án đơn giản chỉ chứa một action mà khi khẩn cầu ta sẽ nhận được trong web như sau:



Trong bài này chúng ta sẽ tạo một dự án có cấu trúc tổ chức như sau

 <pre> SpringMVC ├── JAX-WS Web Services ├── Deployment Descriptor: SpringMVC ├── Java Resources │ ├── src │ │ ├── com.lycato.controller │ │ │ └── WelcomeController.java │ └── Libraries ├── JavaScript Resources ├── build ├── WebContent │ ├── META-INF │ └── WEB-INF │ ├── jsp │ │ └── MyView.jsp │ ├── lib │ │ ├── dispatcher-servlet.xml │ │ └── web.xml │ └── index.jsp </pre>	<ul style="list-style-type: none"> ✓ Web.xml Khai báo cấu hình cần thiết của ứng dụng web ✓ Dispatcher-servlet.xml Khai báo cấu hình tối thiểu về Spring MVC ✓ Index.jsp Trang khởi đầu mặc định của ứng dụng web ✓ Com.lycato.controller.WelcomeController Controller đơn giản ✓ MyView.jsp View đơn giản ✓ Lib/*.jar Các tập tin thư viện cần thiết
---	---

Bước 1: Tạo dự án



Bước 2: Chuẩn bị thư viện

Chép tất cả các tập tin thư viện *.jar vào thư mục WEB-INF/lib

Bước 3: Cấu hình ứng dụng

Bổ sung vào WebContent/WEB-INF 2 tập tin cấu hình có nội dung như hướng dẫn sau

File web.xml

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Web.xml này cấu hình 2 thành phần quan trọng:

- ✓ Chỉ ra index.jsp là trang mặc định, nghĩa là khi gọi đến thư mục thì index.jsp sẽ chạy
- ✓ DispatcherServlet sẽ tiếp nhận tất cả các request kết thúc bởi .htm

File dispatcher-servlet.xml

```
<!-- Khai báo ViewResolver (xử lý View) -->
<bean id="viewResolver" p:prefix="/WEB-INF/jsp/" p:suffix=".jsp"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>

<!-- Cấu hình cho phép sử dụng Spring MVC Annotation -->
<mvc:annotation-driven/>
<context:annotation-config />

<!-- Nơi tìm kiếm các Component -->
<context:component-scan base-package="com.lycato.controller"/>
```

File cấu hình này

- ✓ Chỉ ra các View đặt trong /WEB-INF/jsp/ và bổ sung phần đuôi cho view là .jsp
- ✓ Cho phép sử dụng annotation trong ứng dụng Spring MVC
- ✓ Chỉ ra các component (@Controller, @Service, @Repository, @Component...) sẽ được tìm trong package com.lycato.controller.

Bước 4: Tạo WelcomeController

```
package nhatnghe.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class MyController {
```



```
@RequestMapping(value="myAction", method=RequestMethod.GET)
public String myAction(ModelMap model) {
    model.addAttribute("message", "Welcome to Spring MVC");
    return "MyView";
}
```

Controller này định nghĩa một action có tên MyAction chỉ được phép gọi theo phương thức GET. Action này sẽ đặt vào model một thuộc tính có tên message và trả về "MyView". Với kết quả này thì DispatcherServlet sẽ chuyển về view /WEB-INF/jsp/ MyView.jsp được cấu hình trong dispatcher-servlet.xml.

Bước 5: Tạo View

WEB-INF/jsp/MyView.jsp

```
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC</title>
</head>
<body>
    <h2>${message}</h2>
</body>
</html>
```

Hiển thị giá trị của thuộc tính message được controller đặt vào model trước đó.

Bước 6: Chạy project

Bạn cần tạo index.jsp (tập tin mặc định) với nội dung sau để khẩn cầu action đã được định nghĩa trong Controller.

```
<jsp:forward page="MyAction.htm"/>
```

Dòng mã này chuyển tiếp sang trang MyAction.htm, tức gọi đến action được ánh xạ là MyAction trong Controller nào đó.

Sau đó chạy bằng cách: phải chuột trên project > Run As > Run on Server và bạn sẽ nhận được kết quả như đã mô tả.

5.2 RequestMapping

5.2.1 Ánh xạ action

@RequestMapping(value, method, params) được sử dụng để ánh xạ yêu cầu (Request) với phương thức xử lý Request (Action). Annotation này có thể sử dụng để đánh dấu cho 1 Controller và cho từng Action riêng lẻ.

- ✓ Khi đánh dấu cho 1 Controller thì các thiết lập của nó được áp dụng cho tất cả các Action trong Controller đó.

- ✓ Khi đánh dấu cho từng Action riêng thì chỉ áp dụng các thiết lập cho action đó.

Các thuộc tính của `@RequestMapping` giúp bạn đưa ra nhiều phương án ánh xạ khác nhau dựa vào việc thiết lập các thông số của nó:

- ✓ **value**: chỉ ra action được ánh xạ với phương thức xử lý
- ✓ **method**: chỉ ra phương thức truyền dữ liệu của trình duyệt web (POST, GET, PUT, DELETE...)
- ✓ **params**: chỉ ra tham số (parameter) bắt buộc phải có để thực hiện action

Một phương thức xử lý action sẽ được thực hiện khi các thuộc tính ánh xạ trùng khớp.

```
@Controller
@RequestMapping(value="myAction")
public class MyController {
    /*
     * POST|GET myAction.htm
     */
    @RequestMapping()
    public String myAction1(ModelMap model) {...}

    /*
     * POST myAction/subAction.htm
     */
    @RequestMapping(value="subAction", method=RequestMethod.POST)
    public String myAction2(ModelMap model) {...}

    /*
     * POST|GET myAction.htm?btnInsert
     */
    @RequestMapping(params="btnInsert")
    public String myAction3(ModelMap model) {...}
}
```

Địa chỉ của mỗi action được xác định:

ActionURL = Controller[@RequestMapping(value)] + Method[@RequestMapping(value)] + suffix. Trong đó suffix là phần kết thúc của mẫu url trong web.xml (<url-pattern>*.htm</url-pattern>).

Ngoài ra việc lựa chọn phương thức để thực hiện action còn phụ thuộc vào params và method như đã nói ở trên.

Theo lý giải trên, chúng ta dễ dàng lấy suy luận ra 3 action trong Controller như sau:

Action			Phương thức thực hiện
URL	Phương thức truyền	Tham số	
myAction.htm	POST hoặc GET		myAction1()
		btnInsert	myAction3()
myAction/subAction.htm	POST		myAction2()

5.2.2 Xử lý tham số yêu cầu

5.2.2.1 Sử dụng `HttpServletRequest`

Phương pháp này cho phép bạn nhận tham số yêu cầu như trong lập trình Servlet và JSP. Bạn có thể sử dụng các phương thức sau để xử lý các tham số yêu cầu.

	Phương thức	Mô tả
1	<code>String getParameter(String name)</code>	Nhận giá trị tham số đơn theo tên
2	<code>String[] getParameterValues(String name)</code>	Nhận các giá trị tham số đa giá trị (listbox hoặc các checkbox cùng tên)
3	<code>Enumeration getParameterNames()</code>	Nhận tất cả tên tham số yêu cầu

```
@RequestMapping(value="myAction")
public String myAction(HttpServletRequest request){
    // Nhận tham số có tên txtUserName
    String name = request.getParameter("txtUserName");
}
```

Nếu bạn khẩn cầu action với url `myAction?txtUserName=NguyenNghiem` thì giá trị của biến `name` là sẽ là `NguyenNghiem`.

Với phương pháp này bạn có thể nhận giá trị của cookie và header (sẽ được học sau).

5.2.2.2 Sử dụng `@RequestParam ()`

Phương pháp này cho phép ánh xạ một tham số yêu cầu với một đối số của phương thức action. Điều này đồng nghĩa với việc tự động chuyển đổi kiểu dữ liệu cho phù hợp với đối số của action.

```
@RequestMapping(value="myAction")
public String myAction(
    @RequestParam("ten") String name,
    @RequestParam("luong") double salary){...}
```

Ví dụ này chuyển giá trị các tham số (`ten`, `luong`) vào đối số (`name`, `salary`) đồng thời chuyển đổi sang kiểu phù hợp với đối số (`String`, `double`). Nếu chúng ta khẩn cầu action với url `myAction?ten=ngiem&luong=1000` thì giá trị của đối số `salary` là 1000 và `name` là `ngiem`.

5.2.2.3 Sử dụng `@PathVariable()`

Phương pháp này cho phép ánh xạ một phần của url yêu cầu vào các đối số của action. Nó giúp truyền dữ liệu cho action thông qua địa chỉ url.

```
@RequestMapping(value="{id}/myAction")
public String myAction(
    @PathVariable("id") int studentId){...}
```

Ví dụ này chuyển phần `{id}` trên đường dẫn url vào đối số `masv` đồng thời chuyển sang kiểu `int`. Nếu chúng ta khẩn cầu action với url `2000/myAction.htm` thì giá trị đối số `studentId` sẽ là 2000.

5.2.2.4 Sử dụng JavaBean

Phương pháp này cho phép nhận tất cả các tham số cùng tên với các thuộc tính của bean đồng thời chuyển đổi kiểu tự động cho phù hợp với các giá trị của các thuộc tính trong bean.

```
@RequestMapping(value="myAction") public String myAction(UserInfo user){...}
```

Giả sử lớp `UserInfo` gồm 2 thuộc tính là `id` và `name` như định nghĩa sau

```
package nhatnghe.model;

public class UserInfo {
    int id;
    String password;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Vậy khi bạn nhấn cầu action với url `myAction.htm?id=100&name=NguyenNghiem` thì các thuộc tính `id` và `name` của bean `user` (đối số của phương thức `myAction`) sẽ là 100 và `NguyenNghiem`. Nghĩa là giá trị của các tham số sẽ được chuyển vào các thuộc tính cùng tên của bean và chuyển đổi sang kiểu phù hợp với thuộc tính của bean.

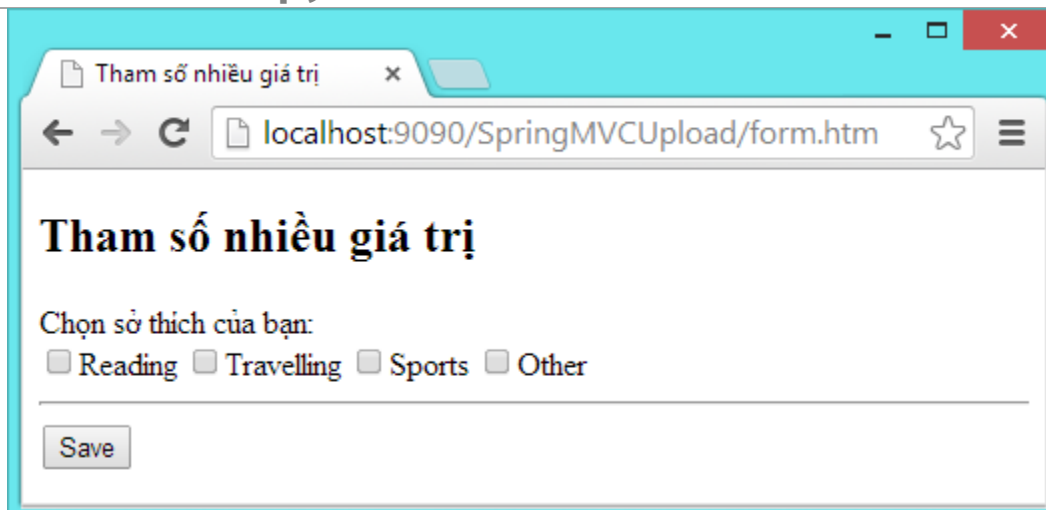
5.2.2.5 Tiếp nhận tham số nhiều giá trị

Khi bạn chọn nhiều mục trong một `ListBox` hay các checkbox cùng tên (chắn hạn như mỗi người có nhiều sở thích hay kiêm nhiều chức danh hay nói được nhiều thứ tiếng...). Giao diện thì quá dễ để xây dựng nhưng vấn đề đặt ra ở đây không phải là giao diện mà lại là cách để nhận các mục chọn trên giao diện.

Chúng ta tìm hiểu sâu hơn vấn đề này qua ví dụ sau.

5.2.2.5.1 Giao diện

Giả sử chúng ta cần nhận các sở thích được lựa chọn bởi người dùng được thiết kế giao diện như sau:



```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Tham số nhiều giá trị</title>
</head>
<body>
    <h2>Tham số nhiều giá trị</h2>
    <form action="save.htm" method="post">
        <div>Chọn sở thích của bạn:</div>
        <label><input type="checkbox" name="hobbies" value="R">Reading</label>
        <label><input type="checkbox" name="hobbies" value="T">Travelling</label>
        <label><input type="checkbox" name="hobbies" value="S">Sports</label>
        <label><input type="checkbox" name="hobbies" value="O">Other</label>
        <hr>
        <input type="submit" value="Save">
    </form>
</body>
</html>
```

Với giao diện này, bạn thấy các checkbox được đặt chung với tên là hobbies và khi nhấn nút [Save] thì action là save.htm.

5.2.2.5.2 Tiếp nhận giá trị lựa chọn

Để tiếp nhận các mục chọn này, bạn có 3 cách để viết mã trong Spring MVC

- ✓ Sử dụng đối số của phương thức action

```
@RequestMapping("/save")
public String save(@RequestParam("hobbies") String[] hobbies) {
    ...
    return "form";
}
```

- ✓ Sử dụng getParameterValues() của HttpServletRequest

```
@RequestMapping("/save")
public String save(HttpServletRequest request) {
    String[] hobbies = request.getParameterValues("hobbies");
    ...
    return "form";
}
```

✓ Sử dụng lớp model

```
@RequestMapping("/save")
public String save(AdvInfo info) {
    String[] hobbies = info.getHobbies();
    ...
    return "form";
}
```

Trong action trên cần lớp AdvInfo trong đối số của nó. Lớp này chứa thuộc tính hobbies và được định nghĩa như sau:

```
public class AdvInfo {
    private String[] hobbies;

    public String[] getHobbies() {
        return hobbies;
    }

    public void setHobbies(String[] hobbies) {
        this.hobbies = hobbies;
    }
}
```

5.2.3 Model & View

Công việc cuối cùng mà một action phải thực hiện là chọn View để hiển thị và chia sẻ dữ liệu với view này. Vì vậy kết quả return của một action thường là ModelAndView. Đối tượng này mang trong mình nó một đối tượng chứa dữ liệu (được gọi là thuộc tính model) và tên của một view để chỉ rõ view cần hiển thị.

View thì cần đúng một cái, nhưng dữ liệu được chia sẻ giữa action và view có thể cần nhiều hơn một đối tượng hoặc không cần nên kết quả của Action khá đa dạng:

- ✓ ModelAndView: mang cả model và view
- ✓ String: chỉ mang view còn model sẽ được chia sẻ tách riêng qua ModelMap
- ✓ Void: không hiển thị kết quả bằng view mà sử dụng mã gửi trực tiếp qua đối tượng HttpServletResponse.

5.2.3.1 Return ModelAndView

Action sau sẽ trả về ModelAndView mang theo view có tên là success và model có tên là user được sinh ra từ lớp AccountInfo. Hơn thế nữa, ModelAndView còn đính kèm một attribute có tên message.

```
@RequestMapping(value="myaction")
public ModelAndView mymethod() {
    AccountInfo model = new AccountInfo();
    model.setEmail("nghiemn@fpt.edu.vn");

    ModelAndView result = new ModelAndView("success", "user", model);
    result.addObject("message", "Đăng ký thành công !");

    return result;
}
```

Ở phía view có thể hiển thị dữ liệu được chia sẻ từ action này theo một số cách sau:

- ✓ Truy xuất và hiển thị email và message trên trang jsp bằng cách sử dụng EL:
 - ✓ `${user.email}`
 - ✓ `${message}`
- ✓ Buộc dữ liệu lên form spring

```
<form:form commandName="user">
    Email: <form:input path="email">
</form:form>
```

5.2.3.2 Return String

Thay vì kết quả của action trả về ModelAndView thì trường hợp này chỉ trả về String chỉ ra tên view muốn hiển thị, còn model và thuộc tính đính kèm được đặt trong ModelMap. Với cách viết này sẽ linh hoạt hơn vì tách rời view và model do đó được sử dụng nhiều hơn.

```
@RequestMapping(value=" myaction")
public String mymethod(ModelMap map) {
    AccountInfo model = new AccountInfo();
    model.setEmail("nghiemn@fpt.edu.vn");

    map.addAttribute("user", model);
    map.addAttribute("message", "Đăng ký thành công !");

    return "success";
}
```


5.2.3.3 Return void

Đôi khi một action chỉ thực hiện một công việc logic nào đó mà không cần có một view để hiển thị kết quả. Trong trường hợp đó action không trả kết quả gì là lựa chọn tốt nhất.

Ví dụ sau đây khi nhấn call action thì bạn sẽ nhận được kết quả hiển thị trên trang web là "Hello World" mà không cần phải xây dựng View.

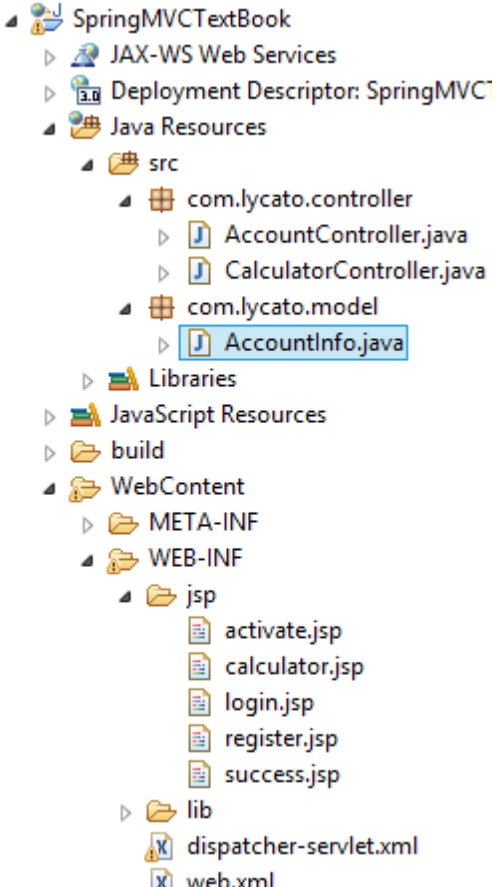
```
@RequestMapping(value=" myaction")
public void mymethod(HttpServletResponse response) throws IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World !");
}
```

5.2.4 Thực hành

Mục tiêu: Kết thúc phần thực hành, bạn có khả năng

- ✓ Định nghĩa Controller với nhiều action khác nhau
- ✓ Phân biệt các action dựa vào parameter

Mô tả: Trong bài này bạn phải tạo một dự án gồm 2 Controller và mỗi Controller chứa vài action.

	<p>AccountController chứa các action để thực hiện việc giả lập đăng nhập, đăng ký và kích hoạt tài khoản.</p> <ul style="list-style-type: none"> ✓ Login.htm ✓ Register.htm ✓ Activate.htm <p>CalculatorController chứa action calculate.htm được nhấn call với các tham số khác nhau là add và sub để thực hiện các phép tính số học đơn giản là cộng và trừ:</p> <ul style="list-style-type: none"> ✓ calculate.htm?add ✓ calculate.htm?sub <p>Các view đặt tại jsp/*.jsp</p> <p>Các file cấu hình</p> <ul style="list-style-type: none"> ✓ Dispatcher-servlet.xml ✓ Web.xml
---	---

Thực hiện: 2 bài thực hành sau đây sẽ giúp bạn đạt được mục tiêu đã đề ra

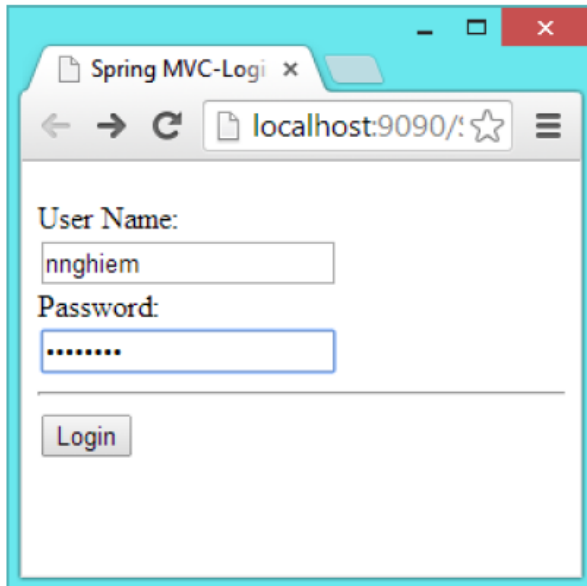
5.2.4.1 Bài 1: Quản lý tài khoản

Trong bài thực hành này, bạn sẽ phải tạo 3 trang chức năng được sử dụng để quản lý tài khoản gồm đăng nhập, đăng ký và kích hoạt tài khoản.

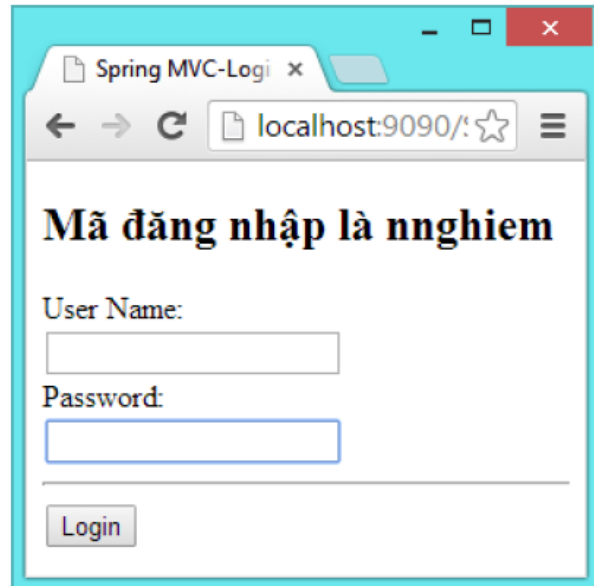
Các trang này chứa các nút chức năng mà khi nhấp chuột vào sẽ khẩn cầu action tương ứng. Nhiệm vụ của bạn là phải định nghĩa các action này để được khẩn cầu một cách phù hợp

Bước 1: Mô tả chức năng

❖ Chức năng đăng nhập

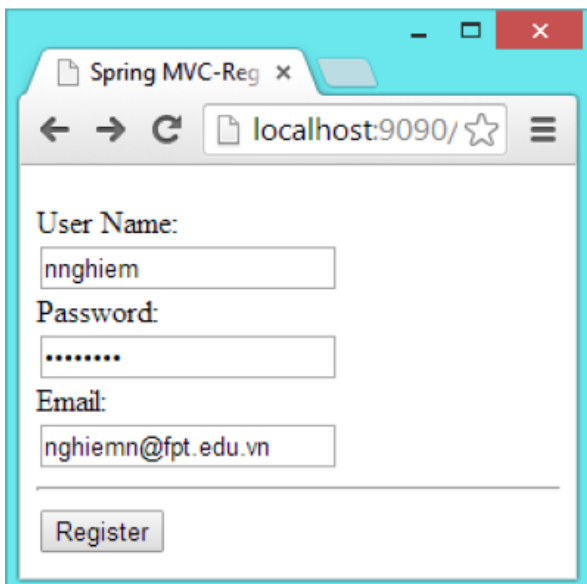


GET: login.htm

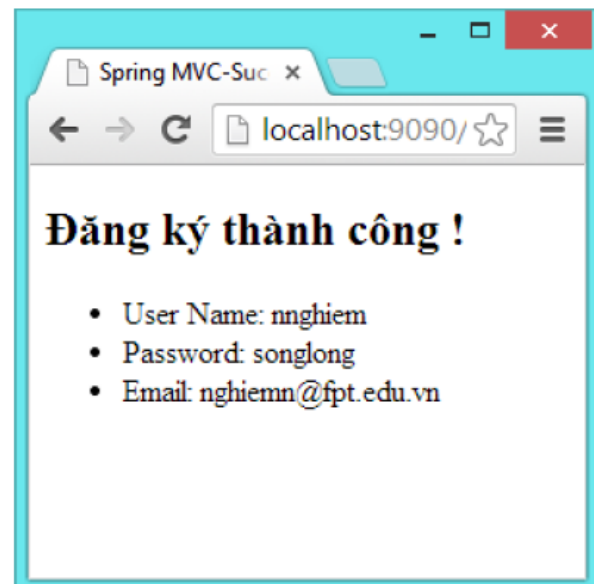


POST: login.htm

❖ Chức năng đăng ký

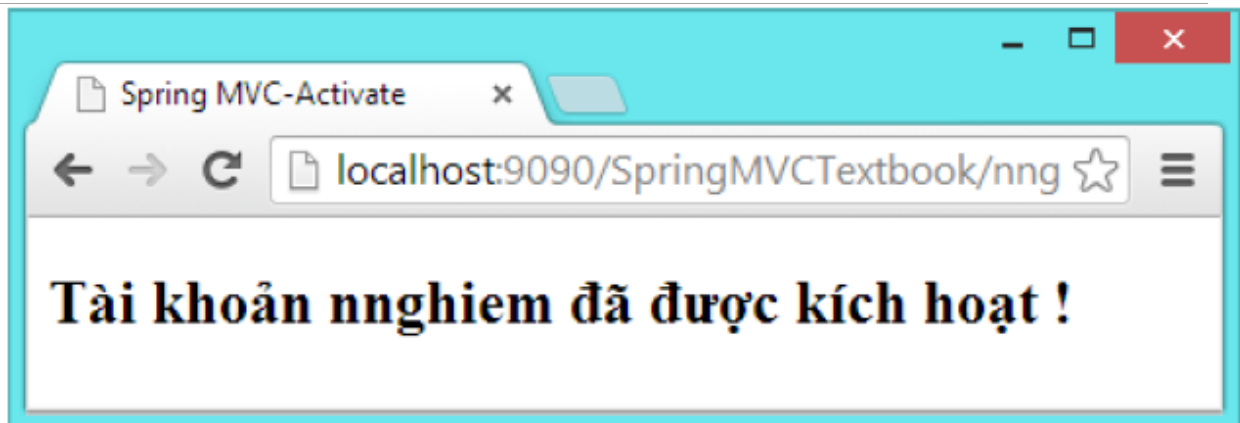


GET: register.htm



POST: register.htm

❖ Kích hoạt tài khoản thành công



GET: `nngiem/activate.htm`

Bước 2: AccountController

AccountController chứa các action được sử dụng để quản lý tài khoản.

```
@Controller
public class AccountController {
    /*
     * GET: login.htm – hiển thị form đăng nhập
     */
    @RequestMapping(value = "login")
    public String login() {
        return "login";
    }

    /*
     * POST: login.htm – thực hiện đăng nhập
     */
    @RequestMapping(value = "login", method = RequestMethod.POST)
    public String login(ModelMap model, @RequestParam("userName") String id,
        @RequestParam("password") String password) {
        model.addAttribute("message", "Mã đăng nhập là " + id);
        return "login";
    }

    /*
     * GET: register.htm – hiển thị form đăng ký
     */
    @RequestMapping(value = "register")
    public String register() {
        return "register";
    }

    /*
     * POST: register.htm – thực hiện đăng ký và hiển thị kết quả
     */
    @RequestMapping(value = "register", method = RequestMethod.POST)
    public String register(ModelMap model, @ModelAttribute("user") AccountInfo info) {
        model.addAttribute("message", "Đăng ký thành công !");
        return "success";
    }

    /*
     * GET: nngiem/activate.htm – thực hiện kích hoạt tài khoản
     */
    @RequestMapping(value = "{id}/activate", method = RequestMethod.GET)
    public String activate(ModelMap model, @PathVariable("id") String id) {
```

```
model.addAttribute("message", "Tài khoản " + id + " đã được kích hoạt!");  
return "activate";  
}  
}
```

Ở trong controller này, bạn thấy rằng mỗi action login.htm và register.htm đều có 2 phiên bản phục vụ cho việc khẩn cầu theo các hình thức GET và POST. Với GET thì chỉ để chuyển sang view chứa form còn POST sẽ là action xử lý.

Riêng activate.htm chỉ có 1 phiên bản GET duy nhất được sử dụng để kích hoạt tài khoản.

Mỗi action trong controller được cài đặt mỗi cách nhận tham số khác nhau:

- ✓ Action login.htm: nhận tham số bằng cách sử dụng đối số của action (@RequestParam)
- ✓ Action register.htm: nhận tham số bằng javabean (@ModelAttribute)
- ✓ Action activate.htm: nhận dữ liệu từ được dẫn url gọi action (@PathVariable)

Bước 3: Tạo AccountInfo

AccountInfo được sử dụng trong action register.htm của AccountController để nhận dữ liệu form đăng ký thành viên bao gồm 3 thuộc tính là id, password và email.

Lớp này phải định nghĩa tuân theo qui ước của JavaBean

- ✓ Phải là public class
- ✓ Phải có constructor không tham số
- ✓ Các thuộc tính phải được định nghĩa dạng getter/setter.
 - Các phương thức setter được sử dụng để ghi dữ liệu vào bean
 - Các phương thức getter được sử dụng để đọc dữ liệu từ bean
 - Muốn lấy tên thuộc tính của bean, bạn chỉ việc bỏ get hoặc set sau đó đổi ký tự phần còn lại sang chữ thường bạn sẽ có tên thuộc tính. Ví dụ bạn có phương thức getSalary(), sau khi bỏ get bạn có Salary và sau khi đổi ký tự đầu (S) sang ký tự thường bạn có salary và đó là thuộc tính của bean. Qui ước này rất quan trọng vì bạn sẽ sử dụng trong JSP dạng EL là \${user.id} nghĩa là trong user có phương thức getId().

```
package nhatnghe.controller;
```

```
public class AccountInfo {  
    private String id;  
    private String password;  
    private String email;  
  
    getters/setters  
}
```

Bước 4: Thiết kế view

- ✓ View login.jsp: Giao diện cho chức năng đăng nhập
- ✓ View register.jsp và success.jsp: giao diện cho chức năng đăng ký
- ✓ View activate.jsp: giao diện cho chức năng kích hoạt tài khoản

View login.jsp gồm 2 phần

- ✓ `{message}` là để hiển thị dòng thông báo được tạo ra từ action `login.htm` `model.addAttribute("message", "...")`. Vì vậy dòng này chỉ hiển thị sau khi đã nhấp chuột vào nút [Login]
- ✓ Form: chứa các trường nhận thông tin đăng nhập gồm id và password. Các trường này sẽ được chuyển vào các đối số id và password của action `login.htm`

```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC-Login</title>
</head>
<body>
    <h2>{message}</h2>
    <form action="login.htm" method="post">
        <div>User Name:</div>
        <input name="userName">
        <div>Password:</div>
        <input name="password" type="password">
        <hr>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

View register.jsp

Cung cấp form chứa thông tin tài khoản gồm id, email và password. Các trường này sẽ được chuyển vào các thuộc tính cùng tên của bean `AccountInfo` của action `register.htm`.

```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC-Register</title>
</head>
<body>
    <form action="register.htm" method="post">
        <div>User Name:</div>
        <input name="id">
        <div>Password:</div>
        <input name="password" type="password">
        <div>Email:</div>
        <input name="email">
        <hr>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

View success.jsp

View này chỉ để hiển thị kết quả thông tin của bean `user` từ model được tạo ra trong action `register.htm` bởi mã lệnh `@ModelAttribute("user")`

```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
```

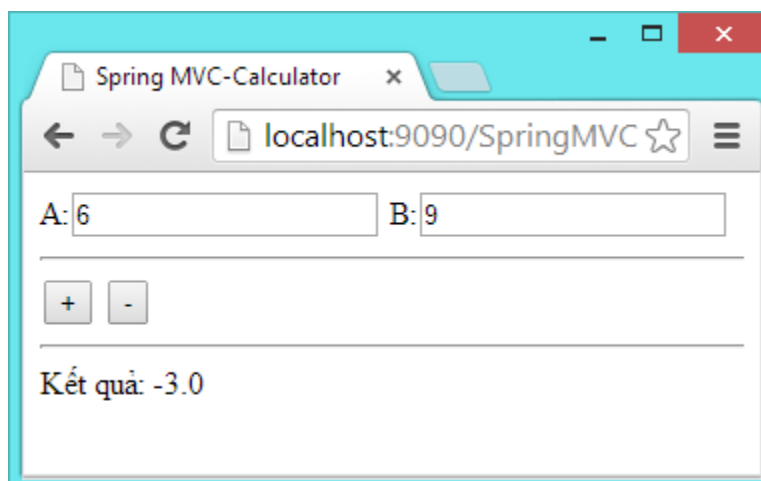
```
<meta charset="utf-8">
<title>Spring MVC-Success</title>
</head>
<body>
    <h2>${message}</h2>
    <ul>
        <li>User Name: ${user.id}</li>
        <li>Password: ${user.password}</li>
        <li>Email: ${user.email}</li>
    </ul>
</body>
</html>
```

View activate.jsp hiển thị dòng thông báo kết quả kích hoạt

```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC-Activate</title>
</head>
<body>
    <h2>${message}</h2>
</body>
</html>
```

5.2.4.2 Bài 2: Máy tính cá nhân

Trong bài này, bạn phải xây dựng máy tính cá nhân thực hiện 2 phép tính cộng và trừ để minh họa việc lựa chọn action bằng cách sử dụng các tham số.



Bước 1: Tạo controller

Controller này chứa action calculate.htm với 3 cách gọi các nhau

- ✓ GET: calculate.htm – Khi cần không tham số
- ✓ POST: calculate.htm?add – Khi cần với tham số add
- ✓ POST: calculate.htm?sub – Khi cần với tham số sub

```
@Controller
@RequestMapping(value = "calculate")
public class CalculatorController {
```

```
/*
 * GET| POST: calculate.htm – Khẩn cầu không tham số
 */
@RequestMapping()
public String calculate() {
    return "calculator";
}

/*
 * POST: calculate.htm?add – Khẩn cầu với tham số add
 */
@RequestMapping(params = "add", method = RequestMethod.POST)
public String add(ModelMap model, @RequestParam("a") double a,
    @RequestParam("b") double b) {
    model.addAttribute("result", a + b);
    return "calculator";
}

/*
 * POST: calculate.htm?sub – Khẩn cầu với tham số sub
 */
@RequestMapping(params = "sub", method = RequestMethod.POST)
public String sub(ModelMap model, @RequestParam("a") double a,
    @RequestParam("b") double b) {
    model.addAttribute("result", a - b);
    return "calculator";
}
}
```

Bước 2: Tạo View

```
<%@page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC-Calculator</title>
</head>
<body>
    <form action="calculate.htm" method="post">
        A:<input name="a" value="${param.a}">
        B:<input name="b" value="${param.b}">
        <hr>
        <input name="add" type="submit" value="+">
        <input name="sub" type="submit" value="-">
        <hr>
        Kết quả: ${result}
    </form>
</body>
```



```
</html>
```

- ✓ Với giao diện này, khi nhấp nút [+] hoặc [-] thì hiển cầu calculate.htm cùng với tham số add hoặc sub tương ứng với nút được nhấp.
- ✓ `${param.a}` và `${param.b}` sẽ hiển thị giá trị của tham số có tên a và b trở lại các ô nhập cùng tên
- ✓ `${result}`: hiển thị giá trị của thuộc tính trong model được tạo bởi action `model.addAttribute("result")`

5.3 CONFIGURATION & DI

5.3.1 Cấu hình ứng dụng Spring MVC

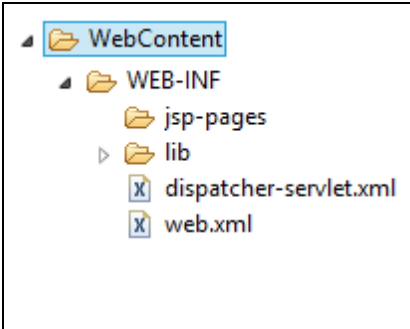
Cấu hình ứng dụng Spring luôn là trái tim của ứng dụng. Môi trường sẽ dựa vào thông tin cấu hình để chuẩn bị và thực hiện theo các yêu cầu được quy ước.

Với Spring bạn có thể sử dụng XML để cấu hình ứng dụng và/hoặc Annotation cũng có thể được sử dụng để thay thế công việc cấu hình này.

Thông thường người ta sử dụng cả 2 loại này để cấu hình ứng dụng. Khi đó Annotation chỉ đóng vai trò giảm tải một số thành phần trong file cấu hình để tránh sự phức tạp khi file cấu hình quá lớn. XML vẫn phải được sử dụng chính để chứa các khai báo cơ bản nhất của hệ thống hoặc những bean củ không sử dụng annotation lúc định nghĩa.

5.3.1.1 Cấu hình bằng XML

File cấu hình ứng dụng

	<p>Trong ứng dụng đơn giản, chúng ta thường cấu hình ứng dụng với 2 file:</p> <ul style="list-style-type: none"> ✓ Web.xml: cấu hình ứng dụng web của java, trong đó có chỉ ra file cấu hình Spring MVC là dispatcher-servlet.xml ✓ Dispatcher-servlet.xml: cấu hình dành riêng cho ứng dụng Spring MVC
---	---

File cấu hình web.xml

File web.xml được sử dụng để cấu hình nhiều công việc khác nhau cho ứng dụng web nói chung (jsp, servlet, struts, spring...). Trong phần này chúng ta chỉ khai thác các khai báo liên quan đến ứng dụng Spring MVC.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <display-name>Ứng dụng web</display-name>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
```

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class> org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
</web-app>
```

Trong khai báo này thì `org.springframework.web.servlet.DispatcherServlet` được sử dụng để tiếp nhận tất cả các yêu cầu có địa chỉ url kết thúc bởi `.htm`. `DispatcherServlet` sẽ phân giải để chuyển đến các action phù hợp với các yêu cầu dựa vào định dạng của url.

Hãy lưu ý rằng, trong file cấu hình này chúng ta sử dụng `dispatcher` để đặt tên cho servlet. Theo quy ước mặc định của Spring thì file cấu hình Spring sẽ có tên là "`<tên servlet>-servlet.xml`". Và vì vậy trong trường hợp này là `dispatcher-servlet.xml`.

Nếu chúng ta đặt tên là `<servlet-name>my-config</servlet-name>` thì file cấu hình Spring MVC sẽ là "`my-config-servlet.xml`".

File Cấu hình dispatcher-servlet.xml

Để hiểu kỹ file cấu hình, chúng ta sẽ phân tích một file cấu hình đơn giản sau đây.

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Declare a view resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp-pages/" p:suffix=".jsp" />

    <!-- Spring MVC Annotation -->
    <mvc:annotation-driven />
    <context:annotation-config />
    <!-- Where to find component -->
    <context:component-scan base-package="com.lycato" />
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- maxUploadSize=10MB -->
        <property name="maxUploadSize" value="10485760"/>
    </bean>

</beans>
```

Tập tin cấu hình này chứa đúng 1 thẻ `<beans>`. Thẻ này chứa các thuộc tính `xmlns:xyz` khá phức tạp.

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    ...
</beans>
```

Các namespace này được khai báo để sử dụng cho các thẻ ở phần cấu hình. Thông thường, chúng ta dùng đến đâu thì khai báo đến đó. Tuy nhiên chúng ta có thể khai báo tất cả những gì cần thiết để chuẩn bị cho các khả năng có thể sử dụng cho ứng dụng sau này cũng được.

Trong trường hợp này `xmlns:tx=http://www.springframework.org/schema/tx` được khai báo để chuẩn bị cho việc sử dụng transaction sau này mà không ảnh hưởng gì cho ứng dụng hiện tại. Nghĩa là chúng ta có thể bỏ 3 dòng sau đây khỏi file cấu hình.

- ✓ `xmlns:tx="http://www.springframework.org/schema/tx"`
- ✓ `http://www.springframework.org/schema/tx`
- ✓ `http://www.springframework.org/schema/tx/spring-tx.xsd`

Các khai báo cần thiết

Có một số thành phần gần như bắt buộc phải khai báo, các thẻ `<bean>` còn lại khi nào muốn sử dụng chúng ta sẽ khai báo sau cũng được.

ViewResolver

Sau khi một action trong controller hoàn thành nhiệm vụ thì nó phải chỉ ra view nào cần được sử dụng để hiển thị giao diện kết quả cho người dùng. Chỉ có tên view được chỉ định trong action còn vị trí đặt view hoặc phần mở rộng của view thì giao lại cho ViewResolver giải quyết.

Ví dụ: trong một controller có định nghĩa 1 action như sau

```
/*
 * GET|POST: calculate.htm
 */
@RequestMapping()
public String calculate() {
    return "calculator";
}
```

Và với khai báo ViewResolver được cấu hình như trên

```
<!--Khai báo xử lý view -->
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp-pages/" p:suffix=".jsp" />
```

Khi đó view /WEB-INF/jsp-pages/calculator.jsp sẽ được lựa chọn để sinh giao diện. Ở đây chỉ ra:

- ✓ viewpath = p:prefix + view_name + p:subfix
- ✓ viewpath = "/WEB-INF/jsp-pages/" + "calculator" + ".jsp"
- ✓ viewpath = "/WEB-INF/jsp-pages/calculator.jsp"

Cho phép sử dụng Annotation

Trong các ứng dụng Spring MVC phát triển từ 3.0+ gần như sử dụng annotation để đơn giản hóa công việc phát triển ứng dụng. Để hệ thống Spring nhận biết điều này bạn cần khai báo trong file cấu hình 2 dòng mã XML sau

```
<!--Khai báo cho phép sử dụng annotation -->
<mvc:annotation-driven />
<context:annotation-config />
```

Với khai báo này, trong Spring MVC bạn có thể đánh dấu các thành phần bằng các annotation:

- ✓ @Controller
- ✓ @RequestMapping
- ✓ @Component
- ✓ @Resource
- ✓ @Service
- ✓ @Autowired
- ✓ @Required
- ✓ ...

Vị trí chứa controller

Khi bạn khẩn cầu 1 action, hệ thống Spring phải truy tìm phương thức ánh xạ mới action đó thông qua @RequestMapping(). Như vậy cần phải biết phương thức của lớp nào được ánh xạ để thực hiện yêu cầu. Khai báo trong tập cấu hình sau sẽ chỉ cho hệ thống biết gói chứa các controller.

```
<!--Khai báo chỉ rõ gói chứa các controller -->
<context:component-scan base-package="com.lycato" />
```

Trong trường hợp này, các lớp controller thuộc gói com.lycato. Nếu có nhiều controller thuộc nhiều gói khác nhau thì phải chỉ rõ các gói bằng dãy phẩy:

```
<!--Khai báo chỉ rõ gói chứa các controller -->
<context:component-scan base-package="package1,package2,package3..." />
```

Ví dụ: sau đây định nghĩa 3 controller thuộc 3 package khác nhau gồm com.lycato.admin, com.lycato.site và system.security.

Controller X, Y, Z

```
package com.lycato.admin;

@Controller
public class XController {...}
```

```
package com.lycato.site;

@Controller
public class YController {...}
```

```
package system.security;  
  
@Controller  
public class ZController {...}
```

Để Spring nhận biết các lớp controller này, chúng ta cần khai báo để chỉ rõ chúng trong file cấu hình như sau:

```
<!--Khai báo chỉ rõ gói chứa các controller -->  
<context:component-scan base-package="com.lycato,system.security" />
```

Khai báo các bean tùy biến

File cấu hình của ứng dụng này có khai báo bean sau

```
<bean id="multipartResolver"  
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  
    <!-- maxUploadSize=10MB -->  
    <property name="maxUploadSize" value="10485760"/>  
</bean>
```

Bean này được khai báo để xử lý việc upload file. Phần khai báo cũng thiết lập giá trị cho thuộc tính maxUploadSize là 10485760 (10MB). Lớp CommonsMultipartResolver được cung cấp bởi Spring, vì vậy khi cần sử dụng, chúng ta chỉ cần khai báo là được (bean này sẽ được tìm hiểu kỹ ngay trong bài này).

Trong bài này để các bạn hiểu hơn, chúng ta sẽ xây dựng bean do tự mình viết ra và sử dụng chúng:

Giả sử chúng ta có 1 lớp gồm 2 thuộc tính (email và password) và một phương thức (exist()) sau

```
package com.lycato.model;  
  
public class UserInfo {  
    private String email;  
    private String password;  
  
    public boolean exist() {  
        return email.equals("nghiemn@fpt.edu.vn");  
    }  
  
    Getters/setters  
}
```

Để sử dụng lớp này trong các controller chúng ta cần phải khai báo trong file cấu hình như sau

```
<bean id="user"  
      class="com.lycato.model.UserInfo">  
    <!-- thiết lập thuộc tính email cho bean -->  
    <property name="email" value="songlong2k@gmail.com"/>  
</bean>
```

Sau đó có thể sử dụng bean này trong controller như sau

```
@Controller  
public class UserController {  
    @Autowired  
    UserInfo user;  
  
    @RequestMapping(value = "login")  
    public String login(ModelMap map) {
```

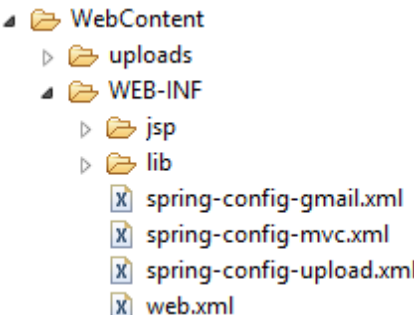
```

        if (user.exist()) {
            return "account";
        }
        map.addAttribute("message", "Invalid email !");
        return "login";
    }
}

```

5.3.1.2 Cấu hình ứng dụng với nhiều file

Trong một ứng dụng Spring MVC quá nhiều cấu hình trên một file sẽ rất khó kiểm soát. Khi đó bạn nên chia thông tin cấu hình thành nhiều file, mỗi file khai báo cấu hình liên quan đến 1 chủ đề nào đó.

	<p>Trong ví dụ này gồm 3 file cấu hình</p> <ul style="list-style-type: none"> ✓ Spring-config-mvc.xml: khai báo cấu hình cơ bản của ứng dụng spring mvc ✓ Spring-config-gmail.xml: khai báo bean gửi email qua google mail ✓ Spring-config-upload.xml: khai báo bean xử lý upload file
---	---

❖ Web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <display-name>SpringMVC</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <!-- Nạp nhiều file cấu hình -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config-*.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>

```

❖ Spring-config-mvc.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <!-- Spring MVC Annotation -->
    <context:annotation-config />

```

```
<mvc:annotation-driven/>

<!-- cấu hình view -->
<bean id="viewResolver"
      p:prefix="/views/" p:suffix=".jsp"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>

<!-- cấu hình controller -->
<context:component-scan base-package="nhatnghe.controller"/>

<bean class="nhatnghe.model.User" id="nghiemn">
  <property name="id" value="nghiemn"/>
  <property name="password" value="iloveyou"/>
</bean>

<bean class="nhatnghe.model.User" id="lththao">
  <property name="id" value="lththao"/>
  <property name="password" value="iloveyou"/>
</bean>
</beans>
```

❖ Spring-config-gmail.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <!-- Spring Mail Sender -->
  <bean id="mailSender"
        class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="smtp.gmail.com" />
    <property name="port" value="465" />
    <property name="username" value="kentphp2@gmail.com" />
    <property name="password" value="songlong" />

    <property name="defaultEncoding" value="UTF-8"/>
    <property name="javaMailProperties">
      <props>
        <prop key="mail.smtp.auth">true</prop>
        <prop key="mail.smtp.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
        <prop key="mail.smtp.socketFactory.port">465</prop>
        <prop key="mail.debug">true</prop>

        <prop key="mail.smtp.starttls.enable">>false</prop>
      </props>
    </property>
  </bean>

</beans>
```

❖ Spring-config-upload.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- maxUploadSize=20MB -->
    <property name="maxUploadSize" value="20485760"/>
  </bean>

</beans>
```


5.3.2 Cấu hình bằng Annotation

5.3.2.1 Xây dựng lớp cấu hình ứng dụng

Lớp này được sử dụng để thay thế các file cấu hình spring

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.lycato")
public class WebConfig {
    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver bean = new InternalResourceViewResolver();
        bean.setPrefix("/WEB-INF/jsp-pages/");
        bean.setSuffix(".jsp");
        return bean;
    }

    @Bean
    public CommonsMultipartResolver multipartResolver() {
        CommonsMultipartResolver bean = new CommonsMultipartResolver();
        bean.setMaxUploadSize(10485760);
        return bean;
    }
}
```

Để dễ hiểu chúng ta lập bảng so sánh giữa 2 phương pháp cấu hình như sau

XML	Annotation
<context:annotation-config />	@Configuration
<mvc:annotation-driven />	@EnableWebMvc
<context:component-scan base-package="" />	@ComponentScan(basePackages="")
<bean id="">	@Bean

5.3.2.2 Các annotation tự khai báo bean

Để lớp bean của bạn được định nghĩa ra tự khai báo với hệ thống, bạn chỉ cần sử dụng một trong các annotation sau đây:

- ✓ @Component([id]): tự khai báo bean với id. Nếu không chỉ định id thì sử dụng tên kiểu để làm id.
- ✓ @Repository(*id+): tương tự @Component(*id+) nhưng chỉ dành để định nghĩa cho các DAO (Data Access Object) làm việc với CSDL thuộc DAL (Data Access Layer) theo mô hình 3-layer.
- ✓ @Service(*id+): tương tự @Component(*id+) nhưng chỉ dành để định nghĩa cho các class thuộc BLL (Business Logic Layer) trong mô hình 3-layer

Ví dụ 1: định nghĩa và khai báo bean có id là userInfo

```
@Component("user")
public class UserInfo {
    private String email;
```

```

private String password;

public boolean exist() {
    return email.equals("nghiemn@fpt.edu.vn");
}

Getters/setters
}

```

Sau đó trong controller có thể nhận biết bằng @Autowire và sử dụng mà không cần phải khai báo bean trong dispatcher-servlet.xml.

```

@Controller
public class UserController {
    @Autowired
    UserInfo user;

    @RequestMapping(value = "login")
    public String login(ModelMap map) {
        if (user.exist()) {
            return "account";
        }
        map.addAttribute("message", "Invalid email !");
        return "login";
    }
}

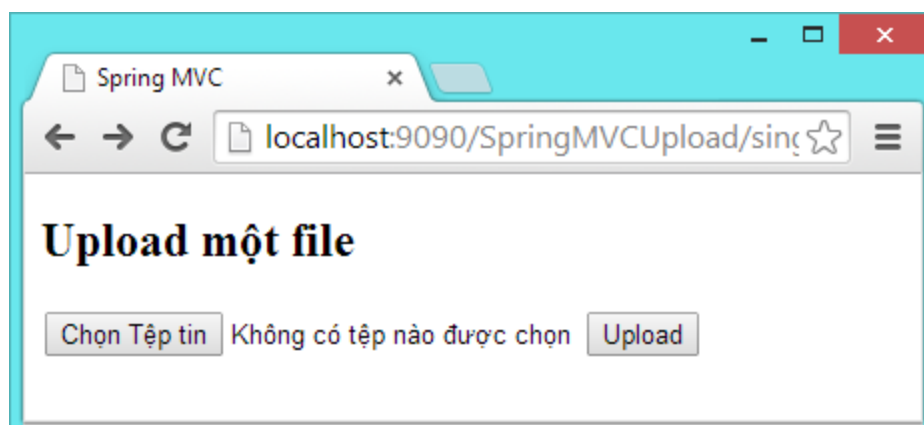
```

5.3.3 Bài thực hành

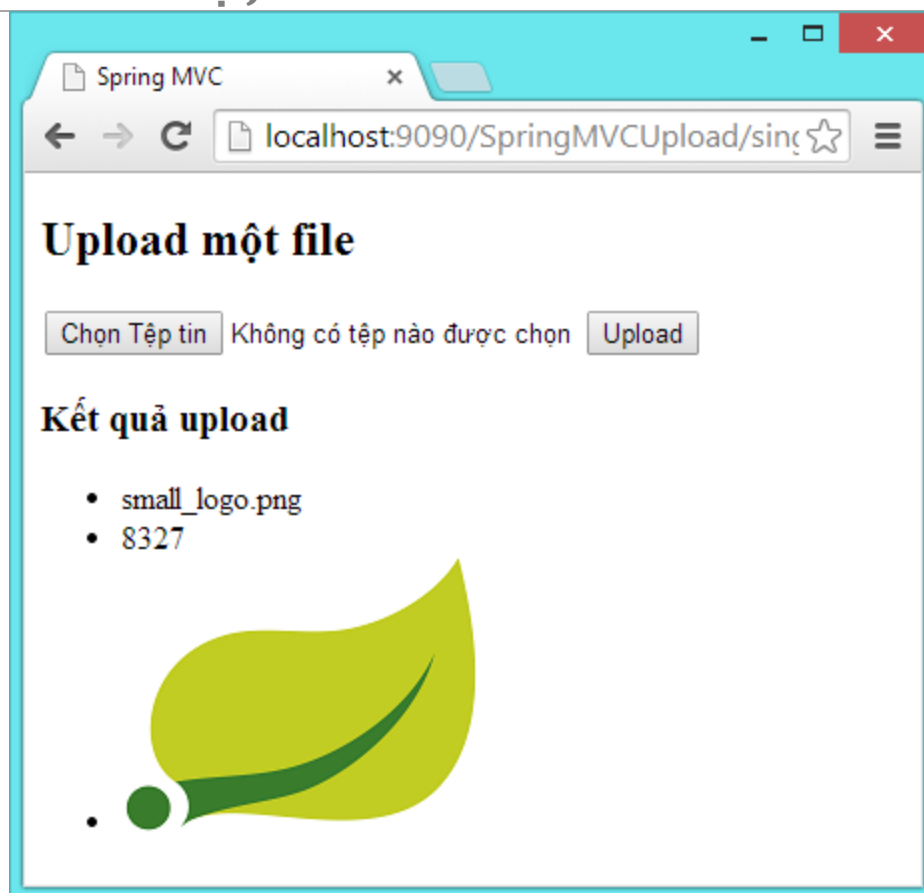
5.3.3.1 Bài 1: Upload một file

Kết thúc bài thực hành, bạn có khả năng

- ✓ Xây dựng ứng dụng upload một file lên server với Spring MVC



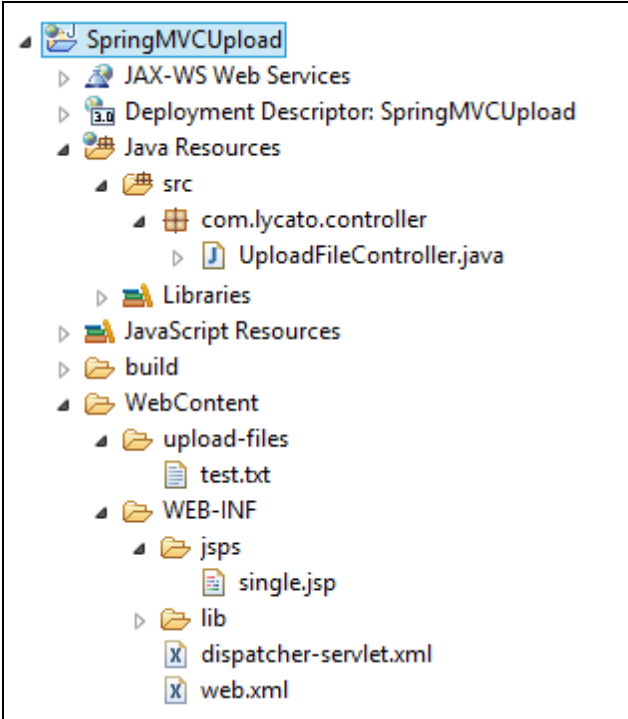
Hình: Trước upload



Hình: Sau upload

Thực hiện

Trong bài này bạn sẽ phải tạo một dự án có cấu trúc như sau

 <pre> SpringMVCUpload ├── JAX-WS Web Services ├── Deployment Descriptor: SpringMVCUpload ├── Java Resources │ ├── src │ │ ├── com.lycato.controller │ │ │ └── UploadFileController.java │ └── Libraries ├── JavaScript Resources ├── build ├── WebContent │ ├── upload-files │ │ └── test.txt │ ├── WEB-INF │ │ ├── jsps │ │ │ └── single.jsp │ │ └── lib │ │ ├── dispatcher-servlet.xml │ │ └── web.xml </pre>	<p>Các thành phần trong dự án gồm</p> <ul style="list-style-type: none"> ✓ UploadFileController.java ✓ Single.jsp ✓ Web.xml ✓ Dispatcher-servlet.xml <p>Sau đây là các bước thực hiện dự án</p>
---	---

UploadFileController chứa 2 action

- ✓ GET: single.htm: hiển thị form upload file
- ✓ POST: single.htm: nhận file upload và lưu vào thư mục
 - `@RequestParam("document") MultipartFile file`: nhận tham số document vào file. `MultipartFile` là kiểu chứa file upload.
 - `File.isEmpty()`: kiểm tra xem có upload file hay không
 - `file.getOriginalFilename()`: lấy tên file upload
 - `file.getSize()`: lấy kích thước file upload
 - `context.getRealPath("upload-files/" + fileName)`: lấy đường dẫn file trong thư mục upload-files của website. Chú ý context là `ServletContext` được tiêm vào ở đầu Controller.
 - `file.transferTo(new File(path))`: chuyển file upload vào thư mục cần thiết.

```
@Controller
public class UploadFileController {
    @Autowired
    ServletContext context;

    /*
     * GET: single.htm
     */
    @RequestMapping(value = "/single", method = RequestMethod.GET)
    public String showUploadFileForm() {
        return "single";
    }

    /*
     * POST: single.htm
     */
    @RequestMapping(value = "/single", method = RequestMethod.POST)
    public String submitUploadFileForm(
        @RequestParam("document") MultipartFile file, ModelMap model)
        throws Exception {
        if (!file.isEmpty()) {
            String fileName = file.getOriginalFilename();
            long fileSize = file.getSize();
            String path = context.getRealPath("upload-files/" + fileName);
            file.transferTo(new File(path));
            model.addAttribute("filename", fileName);
            model.addAttribute("filesize", fileSize);
        }
        return "single";
    }
}
```

Bước 2: single.jsp

Trang jsp gồm 2 phần

- ✓ Form
 - Method=POST
 - Enctype=multipart/form-data
 - `<input type=file>`
- ✓ Kết quả
 - `${filename}`

- `${filesize}`

Kết quả chỉ được hiển thị khi có attribute `${filename}`

```
<%@page pageEncoding="utf-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Spring MVC</title>
</head>
<body>
    <h2>Upload một file</h2>
    <form:form action="single.htm" method="post"
        enctype="multipart/form-data">
        <input type="file" name="document">
        <input type="submit" value="Upload">
    </form:form>
    <c:if test="${!empty filename}">
        <h3>Kết quả upload</h3>
        <ul>
            <li>${filename}</li>
            <li>${filesize}</li>
            <li></li>
        </ul>
    </c:if>
</body>
</html>
```

Bước 3: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>Spring MVC Upload</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

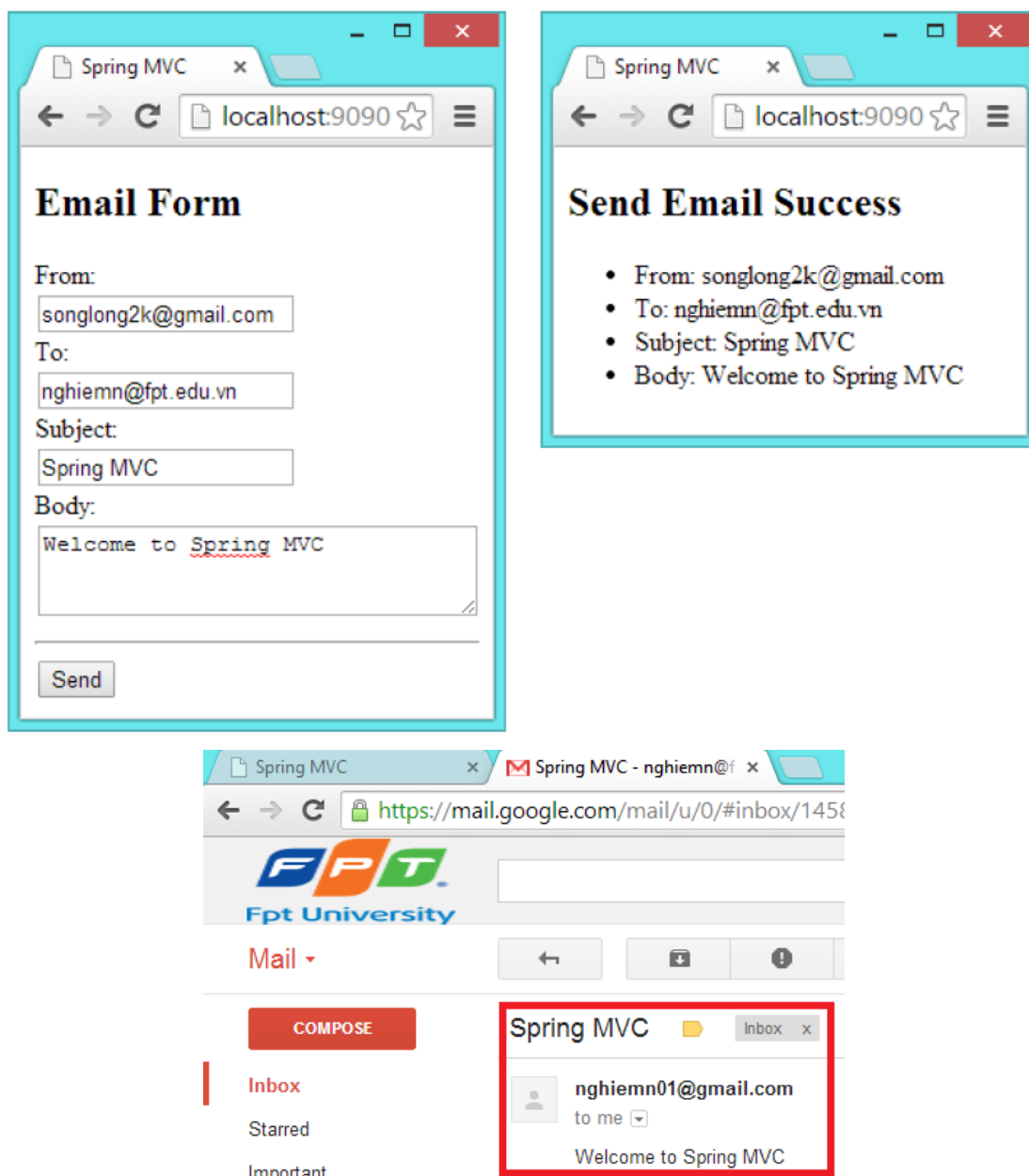
Tên servlet là dispatcher (`<servlet-name>dispatcher</servlet-name>`) nghĩa là tập tin cấu hình sẽ là dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- Declare a view resolver -->
```

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp-pages/" p:suffix=".jsp" />
<!-- Spring MVC Annotation -->
<mvc:annotation-driven />
<context:annotation-config />
<!-- Where to find component -->
<context:component-scan base-package="com.lycato" />
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- maxUploadSize=10MB -->
    <property name="maxUploadSize" value="10485760"/>
</bean>

</beans>
```

5.3.3.2 Bài 2: Send Email đơn giản



Hình: Kết quả nhận email

Bước 1: EmailController.java

Controller này gồm 2 action

- ✓ GET>email.htm: hiển thị form
- ✓ POST>send.htm: nhận thông tin form và gửi email
- ✓ @Autowired JavaMailSender: tiêm bean gửi mail được cấu hình trong spring-config-email.xml

```
@Controller
public class EmailController {
    @Autowired
    JavaMailSender mailSender;

    /**
     * GET: email.htm
     */
    @RequestMapping(value = "email", method = RequestMethod.GET)
    public String showEmailForm(ModelMap model) {
        model.addAttribute("mail", new EmailInfo());
        return "EmailForm";
    }

    /**
     * POST: send.htm
     */
    @RequestMapping(value = "send", method = RequestMethod.POST)
    public String sendEmail(ModelMap model,
        @ModelAttribute("mail") EmailInfo mail) {
        try {
            MimeMessage message = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message, true);
            helper.setFrom(mail.getFrom());
            helper.setTo(mail.getTo());
            helper.setReplyTo(mail.getFrom());
            helper.setSubject(mail.getSubject());
            helper.setText(mail.getBody(), true);
            mailSender.send(message);
        } catch (Exception ex) {
            model.addAttribute("error", ex.getMessage());
            return "EmailForm";
        }
        return "EmailSuccess";
    }
}
```

EmailInfo là lớp model được sử dụng để buộc dữ liệu lên các trường trên form. Lớp này được định nghĩa ở bước ngay sau.

JavaMailSender cung cấp 2 phương thức:

- ✓ createMimeMessage(): tạo mail
- ✓ send(): gửi mail

MimeMessageHelper lớp này cung cấp các phương thức giúp thiết lập các thông tin của một email một cách đơn giản và dễ dàng.

- ✓ setFrom(): email người gửi
- ✓ setTo(): email người nhận

- ✓ setReplyTo(): email nhận reply
- ✓ setSubject(): tiêu đề mail
- ✓ setText(): nội dung email

Bước 2: EmailInfo.java

Lớp này được định nghĩa để mô tả thông tin của một email

```
public class EmailInfo {  
    private String from, to, subject, body;  
  
    getters/setters  
  
}
```

Bước 3: EmailForm.jsp

View này chứa form gửi email. Thuộc tính modelAttribute="mail" chỉ ra command object được sử dụng để buộc đối tượng dữ liệu với form.

```
<%@page pageEncoding="utf-8"%>  
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>  
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>Spring MVC</title>  
</head>  
<body>  
    <h2>Email Form</h2>  
    <form:form action="send.htm" method="post" modelAttribute="mail">  
        <div>From:</div>  
        <form:input path="from" />  
        <div>To:</div>  
        <form:input path="to" />  
        <div>Subject:</div>  
        <form:input path="subject" />  
        <div>Body:</div>  
        <form:textarea path="body" rows="3" cols="30" />  
        <hr>  
        <input type="submit" value="Send">  
    </form:form>  
</body>  
</html>
```

Bước 4: EmailSuccess.jsp

View này dùng để hiển thị thông tin email đã nhập vào.

```
<%@page pageEncoding="utf-8"%>  
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>Spring MVC</title>  
</head>  
<body>  
    <h2>Send Email Success</h2>  
    <ul>  
        <li>From: ${mail.from}</li>  
        <li>To: ${mail.to}</li>
```

```
<li>Subject: ${mail.subject}</li>
<li>Body: ${mail.body}</li>
</ul>
</body>
</html>
```

Bước 5: web.xml

Giá trị của tham số contextConfigLocation có mẫu /WEB-INF/spring-config-*.xml là chỉ ra ứng dụng gồm nhiều file cấu hình bắt đầu bởi spring-config- và kết thúc bởi .xml. Cụ thể trong ứng dụng này có 3 file cấu hình là:

- ✓ Spring-config-gmail.xml: cấu hình bean JavaMailSender
- ✓ Spring-config-mvc.xml: cấu hình cơ bản Spring MVC
- ✓ Spring-config-upload.xml: cấu hình bean xử lý upload file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>

    <display-name>SpringMVC</display-name>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config-*.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

</web-app>
```

Bước 6: spring-config-*.xml

Tạo các file cấu hình sau đây như đã trình bày ở phần cấu hình nhiều file

- ✓ Spring-config-mvc.xml
- ✓ Spring-config-gmail.xml
- ✓ Spring-config-upload.xml

5.4 Spring Form

Spring MVC cung cấp bộ thư viện thẻ form được sử dụng để sinh các phần tử form có buộc với thành phần dữ liệu (command). Nhờ đó việc lập trình điều khiển dữ liệu trên mỗi thành phần của form trở nên đơn giản hơn rất nhiều.

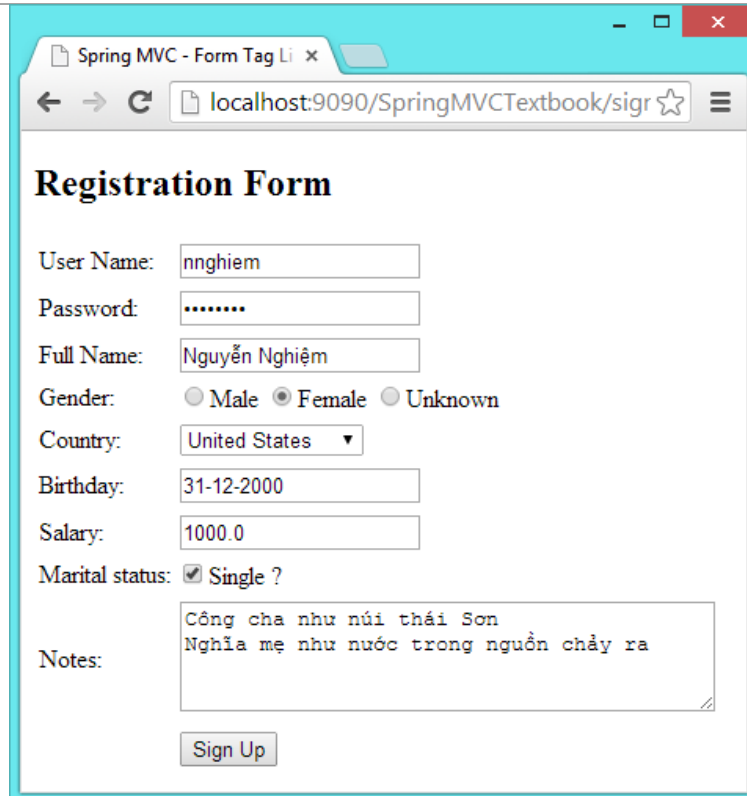
Để thay đổi dữ liệu trên các thành phần giao diện của form, chúng ta chỉ việc lập trình điều khiển giá trị các thuộc tính trên đối tượng command là được.

Spring MVC chỉ hỗ trợ một số thẻ, các thẻ còn lại chúng ta vẫn sử dụng html.

Phần tử	Thẻ <form:>	Thẻ html được sinh ra
Form	<form:form>	<form method="post">
TextBox	<form:input>	<input type="text">
Password	<form:password>	<input type="password">
CheckBox	<form:checkbox>	<input type="checkbox">
RadioButton	<form:radio>	<input type="radio">
Button	<form:button>	<button></button>
Hidden	<form:hidden>	<input type="hidden">
TextArea	<form:textarea>	<textarea>
ComboBox	<form:select>	<select> và <option>
Listbox	<form:select multiple="" size="">	<select multiple size=""> và <option>
CheckBoxList	<form:checkboxes>	Nhiều <input type="checkbox">
RadioButtonList	<form:radios>	Nhiều <input type="radio">
Label	<form:label>	<label>
FileUpload	Không có	<input type="file">
Submit Button	Không có	<input type="submit">
Reset Button	Không có	<input type="reset">
Image Button	Không có	<input type="image">

5.4.1 Form cơ bản

Sử dụng bộ thẻ <form:> để xây dựng giao diện gồm các phần tử form đơn giản như hình sau



Với giao diện này, bạn cần tạo trang jsp có mã như sau. Hãy chú ý các đoạn mã được bôi nền vàng, đó là các thẻ `<form:>` do thư viện form của spring cung cấp.

Trang `signup.jsp`

```
<%@page pageEncoding="utf-8"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC - Form Tag Library</title>
</head>
<body>
    <h2>Registration Form</h2>
    <form:form action="signup.htm">
        <table>
            <tr>
                <td>User Name:</td>
                <td><form:input path="id" /></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><form:password path="password" /></td>
            </tr>
            <tr>
                <td>Full Name:</td>
                <td><form:input path="fullname" /></td>
            </tr>
            <tr>
                <td>Gender:</td>
                <td>
                    <form:radiobutton path="gender" value="0" label="Male" />
                    <form:radiobutton path="gender" value="1" label="Female" />
                    <form:radiobutton path="gender" value="2" label="Unknown" />
                </td>
            </tr>
        </table>
    </form>

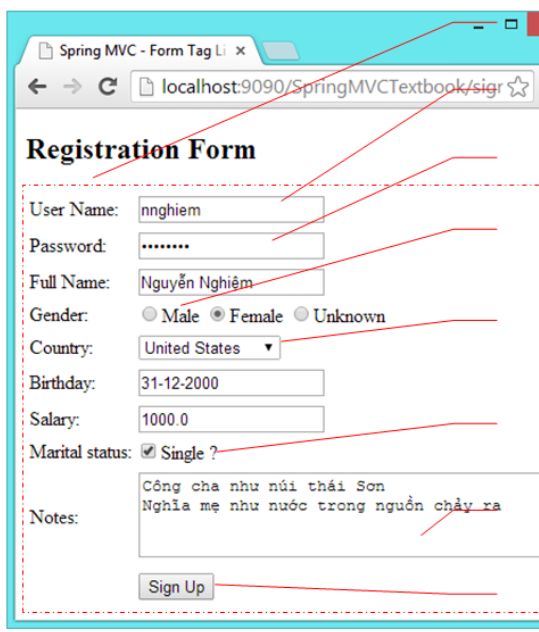
```

```

        </tr>
        <tr>
            <td>Country:</td>
            <td><form:select path="country">
                <form:option value="VN">Vietnam</form:option>
                <form:option value="US">United States</form:option>
                <form:option value="UK">United Kingdom</form:option>
            </form:select></td>
        </tr>
        <tr>
            <td>Birthday:</td>
            <td><form:input path="birthday" /></td>
        </tr>
        <tr>
            <td>Salary:</td>
            <td><form:input path="salary" /></td>
        </tr>
        <tr>
            <td>Marital status:</td>
            <td>
                <label>
                    <form:checkbox path="status" value="true" label="Single ?" />
                </label>
            </td>
        </tr>
        <tr>
            <td>Notes:</td>
            <td><form:textarea path="notes" rows="4" cols="40" /></td>
        </tr>
        <tr>
            <td>&nbsp;</td>
            <td><input type="submit" value="Sign Up" /></td>
        </tr>
    </table>
</form:form>
</body>
</html>

```

Sau đây là hình được tổng hợp lại để chúng ta có một cái nhìn trực quan hơn về các thẻ và các thành phần giao diện. Hình này thay cho lời giải thích của các thẻ.



<form:form action="signup.htm">
<form:input path="id">
<form:password path="password" />
<form:radio button path="gender" value="0" label="Male" />
<form:select path="country"> <form:option value="VN">Vietnam</form:option>
<form:checkbox path="status" value="true" label="Single ?" />
<form:textarea path="notes" rows="4" cols="40" />
<input type="submit" value="Sign Up" />

Để hiển thị được form signup.jsp này, bạn cần một controller chứa một action. Trong action phải cung cấp một đối tượng có tên là command.

Do trong jsp sử dụng <form:form action="signup.htm"> mà không chỉ tên command nên AccountController

```
@Controller
public class AccountController {
    @RequestMapping("/signup")
    public String signup(ModelMap model) {
        model.addAttribute("command", new AccountModel());
        return "signup";
    }

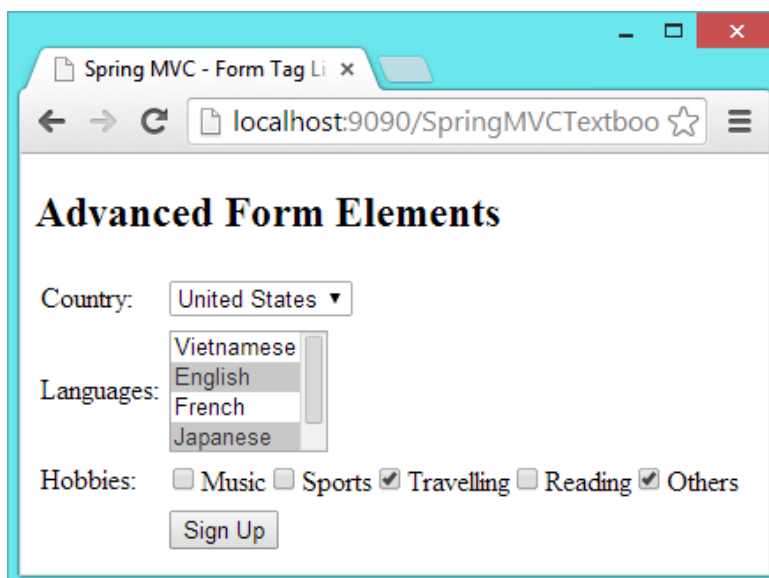
    @RequestMapping(value = "/signup", method = RequestMethod.POST)
    public String signup(ModelMap model,
        @ModelAttribute("command") AccountModel account) {
        return "signup-success";
    }
}
```

AccountModel

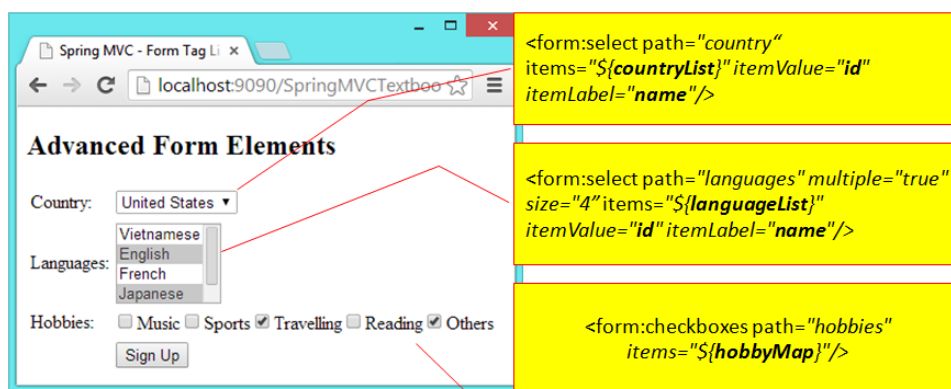
```
public class AccountModel {
    private String id;
    private String password;
    private String fullname;
    private Date birthday;
    private Integer gender = 0;
    private Boolean status;
    private String country;
    private Double salary;
    private String notes;

    getters/setters
}
```

5.4.2 Form nâng cao



```
<%@page pageEncoding="utf-8"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Spring MVC - Form Tag Library</title>
</head>
<body>
    <h2>Advanced Form Elements</h2>
    <form:form action="signup.htm">
        <table>
            <tr>
                <td>Country:</td>
                <td><form:select path="country" items="${countryList}"
                    itemValue="id" itemLabel="name" /></td>
            </tr>
            <tr>
                <td>Languages:</td>
                <td><form:select path="languages" multiple="true" size="4"
                    items="${languageList}" itemValue="id" itemLabel="name" /></td>
            </tr>
            <tr>
                <td>Hobbies:</td>
                <td><form:checkboxes path="hobbies" items="${hobbyMap}" /></td>
            </tr>
            <tr>
                <td>&nbsp;</td>
                <td><input type="submit" value="Sign Up" /></td>
            </tr>
        </table>
    </form:form>
</body>
</html>
```



AccountController

```
@Controller
public class AccountController {
    @RequestMapping("/signup")
    public String signup(ModelMap model) {
        // Cung cấp dữ liệu cho form
        model.addAttribute("command", new AccountModel());
        return "signup ";
    }

    @RequestMapping(value = "/signup", method = RequestMethod.POST)
    public String signup(ModelMap model,
        // Tiếp nhận dữ liệu từ form nhập
```

```

        @ModelAttribute("command") AccountModel account) {
            return "signup-success";
        }

        /*
         * Cung cấp dữ liệu cho ListBox sở thích
         */
        @ModelAttribute("hobbyMap")
        public Map<String, String> populateHobbies() {
            Map<String, String> map = new HashMap<String, String>();
            map.put("0", "Reading");
            map.put("1", "Travelling");
            map.put("2", "Sports");
            map.put("3", "Music");
            map.put("4", "Others");
            return map;
        }

        /*
         * Cung cấp dữ liệu cho CheckBoxList ngôn ngữ
         */
        @ModelAttribute("languageList")
        public List<LanguageModel> populateLanguages() {
            List<LanguageModel> list = new ArrayList<LanguageModel>();
            list.add(new LanguageModel("vi", "Vietnamese"));
            list.add(new LanguageModel("en", "English"));
            list.add(new LanguageModel("fr", "French"));
            list.add(new LanguageModel("ji", "Japanese"));
            list.add(new LanguageModel("ci", "Chinese"));
            return list;
        }

        /*
         * Cung cấp dữ liệu cho ComboBox quốc gia
         */
        @ModelAttribute("countryList")
        public List<CountryModel> populateCountries() {
            List<CountryModel> list = new ArrayList<CountryModel>();
            list.add(new CountryModel("VN", "Vietnam"));
            list.add(new CountryModel("US", "United States"));
            return list;
        }
    }
}

```

AccountModel

```

public class AccountModel {
    private String country;
    private Integer[] hobbies;
    private String[] languages;

    getters/setters
}

```

CountryModel

```

public class CountryModel {
    private String id;
    private String name;

    public CountryModel() {
    }
}

```



```

public CountryModel(String id, String name) {
    this.id = id;
    this.name = name;
}

Getters/setters
}

```

LanguageModel

```

public class LanguageModel {
    private String id;
    private String name;

    public LanguageModel(String id, String name) {
        this.id = id;
        this.name = name;
    }

    Getters/setters
}

```

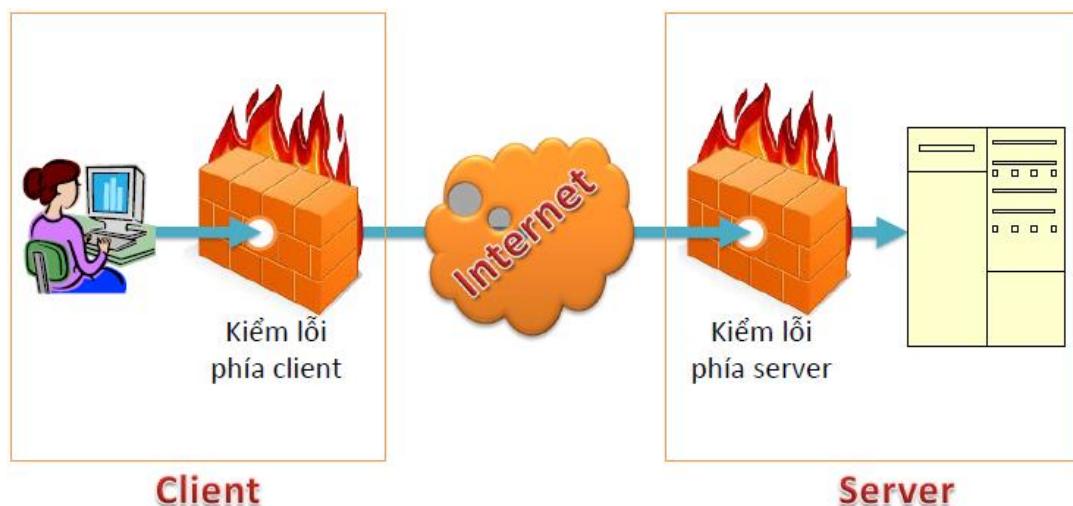
5.5 VALIDATION

Kiểm soát tính hợp lệ dữ liệu đầu vào luôn là vấn đề quan trọng hàng đầu của mỗi lập trình viên. Việc làm này cần được tiến hành ở cả 2 phía là client và server.

Kiểm lỗi phía client để có các phản ứng nhanh cho người dùng, làm cho ứng dụng thân thiện hơn với người dùng.

Kiểm lỗi phía server được thực hiện với những điều kiện không thể thực hiện được trên client. Kiểm lỗi phía server có tính bảo mật cao hơn nhưng thực hiện chậm gây phiền hà cho người dùng. Để khắc phục nhược điểm này người ta sử dụng kỹ thuật ajax để kết hợp kiểm lỗi phía server nhưng phản hồi ngay tức thì cho người dùng.

Ngoài ra với các ứng dụng chặt chẽ cần phải kiểm tra cả 2 phía vì hacker có thể vô hiệu việc kiểm lỗi ở client bằng nhiều cách và nếu không có kiểm lỗi phía server thì ứng dụng của chúng ta sẽ có lỗi.

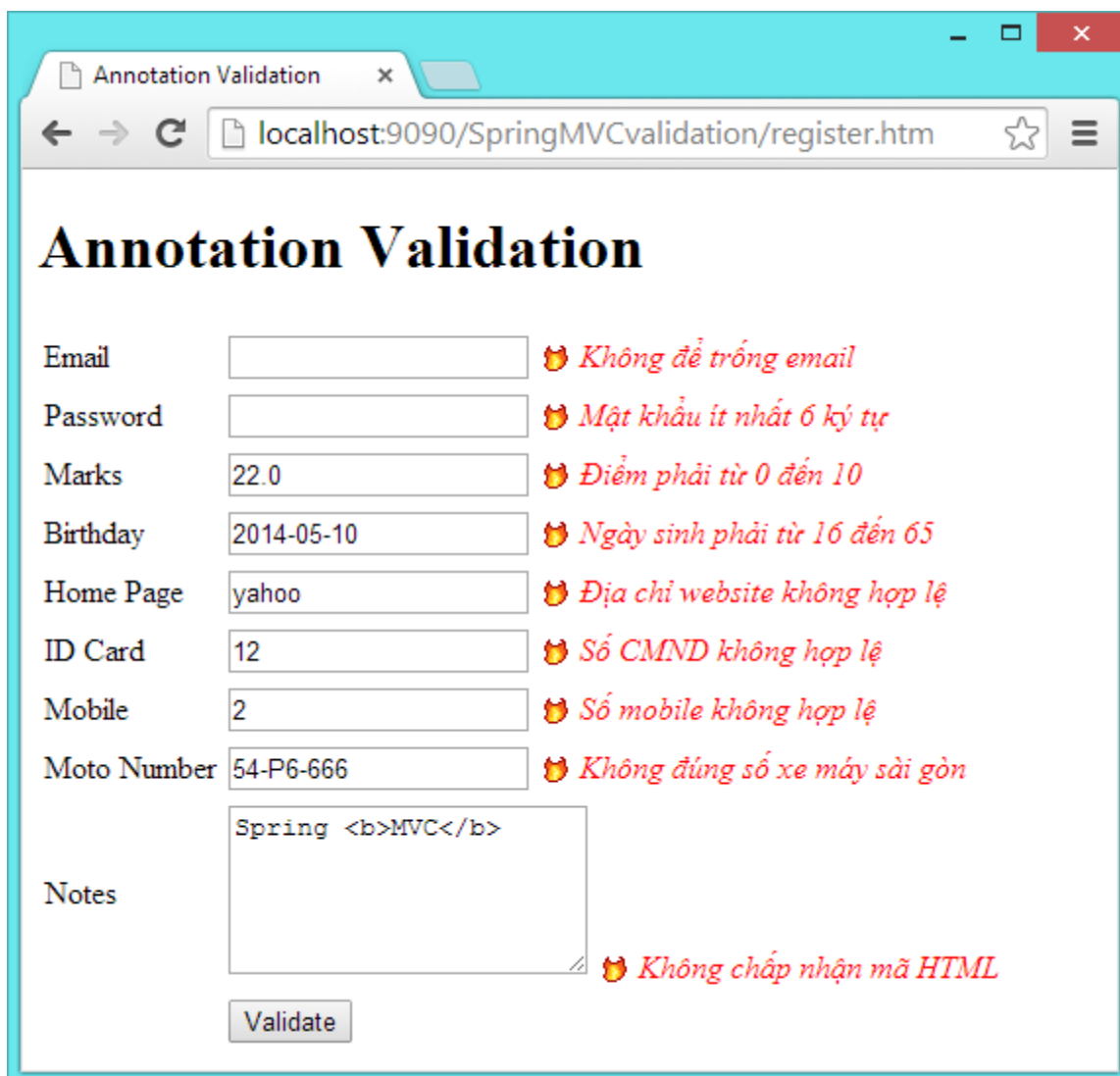


5.5.1 Validation phía client

Kiểm duyệt dữ liệu form nhập trước khi chuyển dữ liệu đến server để xử lý là nhiệm vụ cực kỳ quan trọng. Rất nhiều tiêu chí được đặt ra để kiểm duyệt tùy thuộc vào yêu cầu cụ thể của form nhập liệu. Dù sao vẫn có thể liệt kê một số tiêu chí kiểm duyệt chung chung như sau:

- ✓ Không cho để trống ô nhập...
- ✓ Dữ liệu nhập vào phải theo một khuôn dạng nhất định nào đó: email, creditcard, url...
- ✓ Dữ liệu phải nhập vào phải đúng kiểu: số nguyên, số thực, ngày giờ...
- ✓ Dữ liệu nhập vào phải có giá trị tối thiểu, tối đa, trong phạm vi...
- ✓ Dữ liệu nhập phải đúng theo một kết quả tính toán riêng của bạn...

Sau đây là một ví dụ về việc kiểm lỗi form đăng ký thành viên



The screenshot shows a web browser window with the address bar displaying `localhost:9090/SpringMVCvalidation/register.htm`. The page title is "Annotation Validation". The form contains the following fields and validation messages:

Field	Value	Validation Message
Email		Không để trống email
Password		Mật khẩu ít nhất 6 ký tự
Marks	22.0	Điểm phải từ 0 đến 10
Birthday	2014-05-10	Ngày sinh phải từ 16 đến 65
Home Page	yahoo	Địa chỉ website không hợp lệ
ID Card	12	Số CMND không hợp lệ
Mobile	2	Số mobile không hợp lệ
Moto Number	54-P6-666	Không đúng số xe máy sài gòn
Notes	Spring MVC	Không chấp nhận mã HTML

At the bottom of the form is a "Validate" button.

5.5.1.1 Kiểm lỗi đơn giản với JQuery

Bây giờ chúng ta hãy khám phá khả năng kiểm duyệt dữ liệu đầu vào của JQuery.

Ví dụ sau đây kiểm tra ô nhập "Name" phải nhập ít nhất 3 ký tự và ô nhập "Age" phải nhập số từ 25 đến 65.

```
<%@ page encoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <script src="js/jquery.js"></script>
    <script src="js/jquery.validate.js"></script>
    <script>
        $(function () {
            $("#form1").validate({
                rules: {
                    txtName: { required: true, minlength: 3 },
                    txtAge: { required: true, digits: true, range: [25, 65] }
                },
                messages: {
                    txtAge: { digits: "Nhập số !" },
                    txtName: { required: "Không để trống !",
                        minlength: "Ít nhất 3 ký tự !" }
                },
                errorLabelContainer: "#myError",
                wrapper: "li",
                submitHandler: function (form) {
                    if (confirm("Dữ liệu form đã hợp lệ. Bạn có muốn submit không ?")) {
                        form.submit();
                    }
                }
            });
        });
    </script>
    <style type="text/css">
        label.error{color:Red;}
        input.error { background-color:Red; color:yellow;}
    </style>
</head>
<body>
    <form id="form1" method="post" action="validate.htm">
        <p>Name: </p>
        <input name="txtName" type="text" id="txtName" />
        <p>Age: </p>
        <input name="txtAge" type="text" id="txtAge" />
        <p></p>
        <input type="submit" name="Submit" value="Submit" />
    </form>
    <div id="myError"></div>
</body>
</html>
```

Chạy ứng dụng với các tình huống sau

✓ Để trống các ô và nhấn nút Submit

Name:

Age:

- Không để trống !
- This field is required.

✓ Nhập dữ liệu không hợp lệ và nhấn nút Submit

Name:

Age:

- Ít nhất 3 ký tự !
- Nhập số !

Nhập tuổi không liệu hợp lệ và nhấn nút Submit

Name:

Age:

- Please enter a value between 25 and 65.

Diễn giải:

Thư viện cần thiết cho việc bẫy lỗi:

```
<script src="js/jquery.js"></script>
<script src="js/jquery.validate.js"></script>
<script>
$(function () {
    $("#form1").validate({
        rules: {
            Khai báo các trường cần kiểm lỗi
        },
        messages: {
            Định nghĩa các thông báo lỗi
        },
        errorLabelContainer: Chỉ ra thẻ chứa lỗi,
        wrapper: Chỉ ra thẻ bọc mỗi thông báo lỗi,
        submitHandler: function (form) {
            Mã điều khiển khi form đã nhập đúng
        }
    });
});
</script>
```

5.5.1.2 Danh sách luật kiểm lỗi trong JQuery

Luật	Mô tả	Ví dụ
------	-------	-------

required	Bắt buộc nhập	required:true
required	Bắt buộc nhập nếu tập kết quả của selector rỗng	required:"#chkHoby:blank"
required	Bắt buộc nhập nếu kết quả trả về có giá trị false.	required: function(){return true;}
email	Định dạng email	email:true
url	Định dạng url	url:true
date	Định dạng ngày javascript	date:true
number	Số thực	number:true
digits	Số nguyên	digits:true
creditcard	Định dạng creditcard	creditcard:true
minlength	Số ký tự tối thiểu	minlength:10
maxlength	Số ký tự tối đa	maxlength:100
rangelength	Số ký tự từ min đến max	rangelength:[10, 100]
min	Giá trị tối thiểu	min:10
max	Giá trị tối thiểu	max:100
range	Giá trị từ min đến max	range:[10,100]
equalTo	So sánh giá trị của phần tử và giá trị của selector	equalTo:"#txtPassword"
remote	Hợp lệ khi kết quả kiểm tra từ xa là false.	remote: "remote.htm"

Chú ý: bạn có 2 cách để khai báo luật bắt lỗi

1. Khai báo trong tùy chọn rules như trong ví dụ trên trên
2. Khai báo ngay trong thẻ bạn muốn bắt lỗi

Ví dụ để kiểm lỗi cho ô nhập txtAge của ví dụ trên, bạn có thể khai báo ngay trên thẻ `<input>` như sau:

```
<input class="required digits" min="25" max="65" id="txtAge" />
```

5.5.1.3 Luật kiểm lỗi do người dùng định nghĩa

Trên đây chỉ là danh sách các luật phổ thông hàng ngày. Bạn có thể có những qui luật riêng của mình mà chỉ có bạn mới có thể hiểu và định nghĩa được. Vì vậy JQuery cung cấp cho bạn một cách định nghĩa các luật mới của riêng mình. Hãy xem và phân tích ví dụ sau để hiểu rõ cách để định nghĩa một luật mới.

```

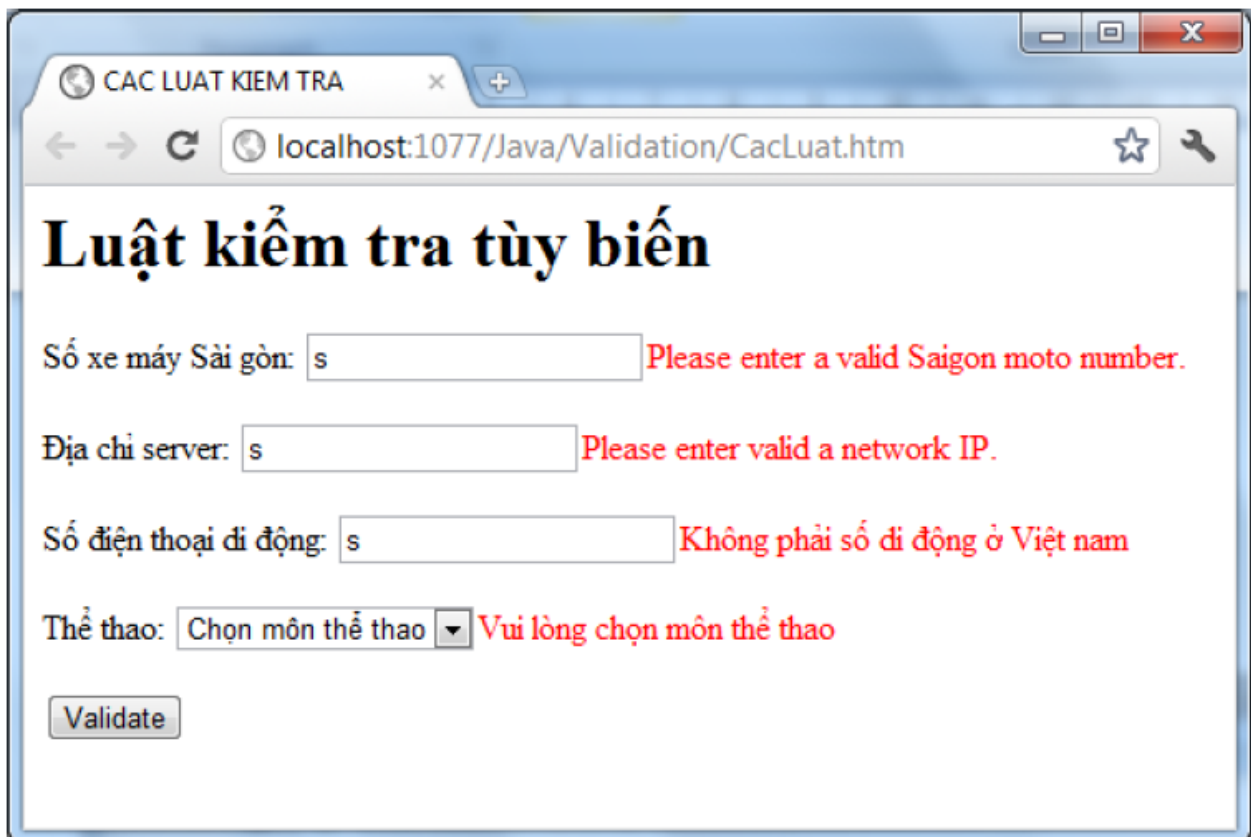
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <script src="jquery.js"></script>
    <script src="jquery.validate.js"></script>
    <script>
        /*--Định nghĩa hàm kiểm tra số di động việt nam--*/
        function fnValidateMobile(value, element) {
            var regex = /^0[0-9]{9,10}$/g;
            return this.optional(element) || regex.test(value);
        }
        /*--Định nghĩa hàm kiểm tra số xe gắn máy sài gòn--*/
        function fnValidateSaigonMoto(value, element) {
            var regex = /^5\d-[A-Z]\d-\d{4}$/g;
            return this.optional(element) || regex.test(value);
        }
        /*--Định nghĩa hàm kiểm tra IP mạng máy tính--*/
        function fnValidateNetworkIP(value, element) {
            var regex = /^(\d{3}\.){3}\d{3}$/g;
            if (this.optional(element) || regex.test(value)) {
                var nums = value.split(".");
                for (var i = 0; i < nums.length; i++) {
                    if (parseInt(nums[i]) > 255) {
                        return false;
                    }
                }
            }
            else {
                return false;
            }
            return true;
        }
        /*--Định nghĩa hàm kiểm tra mục chọn của combo box--*/
        function fnValidateSelectOne(value, element) {
            return (element.value != "none");
        }
        /*--Định nghĩa luật kiểm tra kết hợp với hàm và một thông
        báo lỗi nếu kết quả trả về của hàm có giá trị false--*/
        $.validator.addMethod("selectone",
            fnValidateSelectOne, "Please select an item.");
        $.validator.addMethod("vinaphone",
            fnValidateMobile, "Please enter a valid VinaPhone number.");
        $.validator.addMethod("saigonmoto",
            fnValidateSaigonMoto, "Please enter a valid Saigon moto number.");
        $.validator.addMethod("networkip",
            fnValidateNetworkIP, "Please enter valid a network IP.");
    </script>
    <script type="text/javascript">
        $(function () {
            $("#form1").validate({
                rules:{
                    sport: { selectone: true },
                    mobile: { vinaphone: true }
                },
                messages:{
                    sport: { selectone: "Vui lòng chọn môn thể thao" },
                    mobile: { vinaphone: "Không phải số di động ở Việt nam" }
                }
            });
        });
    </script>

```

```

<style type="text/css">
    label.error{
        color: Red;
    }
</style>
</head>
<body>
    <h1>Luật kiểm tra tùy biến</h1>
    <form id="form1">
        <p>Số xe máy Sài gòn:
        <input type="text" name="moto" class="required saigonmoto">
        <p>Địa chỉ server:
        <input type="text" name="ip" class="networkip">
        <p>Số điện thoại di động:
        <input type="text" name="mobile">
        <p>Thể thao:
        <select name="sport">
            <option value="none">Chọn môn thể thao</option>
            <option value="baseball">Bóng chày</option>
            <option value="basketball">Bóng rổ</option>
            <option value="volleyball">Bóng chuyền</option>
            <option value="football">Bóng đá</option>
        </select>
        <p><input class="submit" type="submit" value="Validate">
    </form>
</body>
</html>

```



Luật kiểm tra tùy biến

Số xe máy Sài gòn: Please enter a valid Saigon moto number.

Địa chỉ server: Please enter valid a network IP.

Số điện thoại di động: Không phải số di động ở Việt nam

Thể thao: Vui lòng chọn môn thể thao

Trong bài trên chúng ta định nghĩa 4 luật kiểm tra mới là vinaphone, saigonmoto, networkip và selectone. Và sau đó áp dụng để kiểm tra dữ liệu cho các thành phần giao diện trên form. Để hiểu được cơ chế định nghĩa và sử dụng chúng ta cần thực hiện các bước sau.

Bước 1: Định nghĩa. cần 2 bước là viết hàm kiểm tra và khai báo luật kiểm với JQuery

✓ Viết hàm kiểm tra:

```
/*--Định nghĩa hàm kiểm tra số di động việt nam--*/  
function fnValidateMobile(value, element) {  
    var regex = /^0[0-9]{9,10}$/g;  
    return this.optional(element) || regex.test(value);  
}
```

Cú pháp của hàm này phải nhận 2 tham số vào là value (giá trị nhập vào) và element (phần tử gây lỗi). Hàm này phải trả về kết quả true (đã hợp lệ) hoặc false (không hợp lệ). Bạn có thể phân tích giá trị để thực hiện kiểm tra nhờ vào tham số value và thay đổi css hay giá trị của phần tử này thông qua tham số element.

✓ Khai báo luật kiểm với JQuery

Sau khi đã định nghĩa hàm kiểm tra, bước tiếp theo là định nghĩa luật kiểm tương ứng với hàm trên và tất nhiên cung cấp thông báo lỗi.

```
/*--Định nghĩa luật kiểm tra kết hợp với hàm và một thông  
báo lỗi nếu kết quả trả về của hàm có giá trị false--*/  
$.validator.addMethod("vinaphone", fnValidateMobile, "Please enter a valid VinaPhone  
number.");
```

Sử dụng phương thức \$.validator.addMethod(rule, method, message) để khai báo luật kiểm. Tham số rule ("vinaphone") là tên luật mới, tham số method ("fnValidateMobile") là tên phương thức kết hợp với luật mới và message ("Please enter a valid VinaPhone number.") là thông báo lỗi.

Bước 2: sử dụng

Bạn sử dụng các luật mới như các luật đã định nghĩa sẵn trong JQuery. Cụ thể là bạn có thể chỉ định trong tùy chọn rules (rules: {mobile: { vinaphone: true }}) của phương thức validate hoặc chỉ ra trên thẻ cần kiểm tra "<input type="text" id="moto" name="moto" class="required saigonmoto">".

5.5.2 Validation phía server

Công việc này xảy ra phía server để kiểm soát dữ liệu gửi đến từ client có hợp lệ hay không. Kiểm lỗi phía server có thể được thực hiện bằng 2 phương pháp

- ✓ Kiểm lỗi bằng tay: tự viết mã để kiểm lỗi
- ✓ Kiểm lỗi bằng annotation: sử dụng các annotation kiểm loại chuyên dụng được cung cấp sẵn.

Một phương thức action để có thể thực hiện kiểm lỗi (cả bằng tay hay annotation) phải khai báo thêm đối số BindingResult.

```
@RequestMapping(method = RequestMethod.POST)
```



```
public String simpleValidate(ModelMap map, BindingResult result,
    @ModelAttribute("user") @Valid UserInfo user) {
    ...
    if(result.hasErrors()){
        return "input-page";
    }
    return "success-page";
}
```

Giả sử chúng ta muốn kiểm lỗi dữ liệu trong đối số user (nhận từ form). Nếu sử dụng phương pháp kiểm lỗi bằng tay thì hãy đặt mã kiểm lỗi tại dấu "...". Còn trong trường hợp kiểm lỗi bằng annotation thì hãy thêm @Valid vào trước đối số UserInfo, cụ thể: @ModelAttribute("user") @Valid UserInfo user.

Để hiểu rõ hơn, chúng ta xét 2 trường hợp riêng biệt thông qua ví dụ.

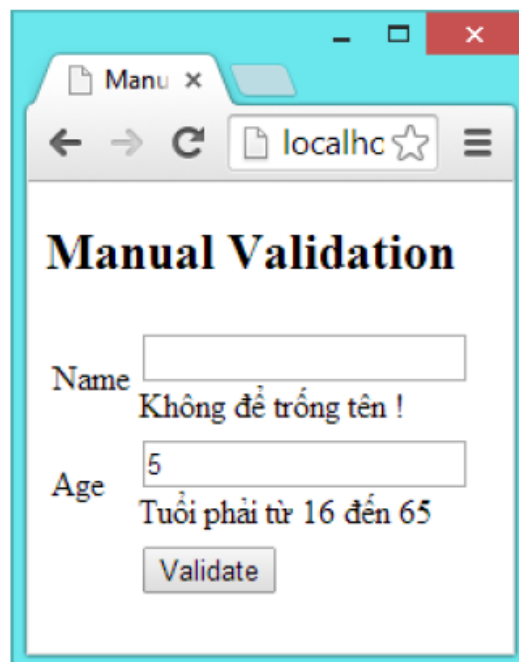
5.5.2.1 Kiểm lỗi bằng tay

Kịch bản của việc kiểm lỗi được mô tả như 2 hình sau. Chạy ứng dụng để hiển thị form gồm 2 ô nhập là name và age.

- ✓ Để form trống và nhấp chuột [Validate]: nhận thông báo yêu cầu nhập
- ✓ Nhập số 5 vào Age và nhấp chuột [Validate]: nhận thông báo tuổi không hợp lệ

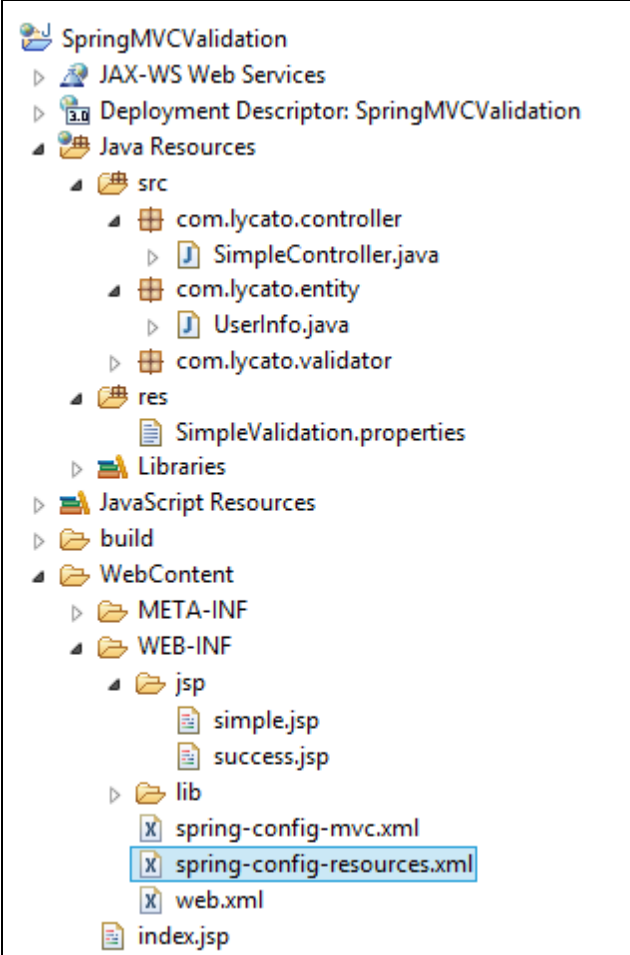


Hình: Để trống và nhấp nút



Hình: Nhập 5 vào Age và nhấp nút

Để thực hiện ứng dụng trên, bạn cần tạo một project có cấu trúc và các thành phần sau

	<p>Các thành phần</p> <ul style="list-style-type: none"> ✓ Simple.jsp, success.jsp: giao diện ✓ SimpleController.java: Controller kiểm lỗi ✓ UserInfo.java: command object buộc thông tin với form ✓ SimpleValidation.properties: định nghĩa thông báo lỗi ✓ *.xml: các file cấu hình <p>Các bước thực hiện</p> <p>Bước 1: UserInfo.java</p> <p>Bước 2: Simple.jsp</p> <p>Bước 3: SimpleController.java</p> <p>Bước 4: SimpleValidation.properties</p> <p>Bước 5: spring-config-resources.xml và *.xml</p> <p>Bước 6: Chạy</p>
--	---

❖ **Bước 1: Định nghĩa command object để buộc dữ liệu với form nhập**

```
public class UserInfo {
    String name;
    Integer age;

    Getters/setters
}
```

❖ **Bước 2: Xây dựng giao diện**

Giao diện gồm 1 form 2 ô nhập có bố trí thông báo lỗi

- ✓ Command object có tên user
- ✓ Hiển thị thông báo lỗi với <form:errors path="name"/>

```
<%@page pageEncoding="utf-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Manual Validation</title>
</head>
<body>
    <h2>Manual Validation</h2>
```

```

<form:form action="simple.html" modelAttribute="user">
    <table>
        <tr>
            <td>Name</td>
            <td><form:input path="name" /> <form:errors path="name" /></td>
        </tr>
        <tr>
            <td>Age</td>
            <td><form:input path="age" /> <form:errors path="age" /></td>
        </tr>
        <tr>
            <td>&nbsp;</td>
            <td><input type="submit" value="Validate" /></td>
        </tr>
    </table>
</form:form>
</body>
</html>

```

Success.jsp

View này hiển thị kết quả sau khi đã nhập đúng dữ liệu

```

<%@ page pageEncoding="utf-8"%>
<html>
<head>
<meta charset="utf-8" />
<title>Validation Success</title>
</head>
<body>
    <h1>Validation Success..!</h1>
    <h3>You Entered</h3>
    <table>
        <tr>
            <td>Name</td>
            <td>:${user.name}</td>
        </tr>
        <tr>
            <td>Age</td>
            <td>:${user.age}</td>
        </tr>
    </table>
</body>
</html>

```

❖ Bước 3: Xây dựng Controller

Controller này chứa action có kiểm lỗi dữ liệu vào của form

```

@Controller
@RequestMapping("simple")
public class SimpleController {
    @RequestMapping
    public String simpleValidate(ModelMap map) {
        map.addAttribute("user", new UserInfo());
        return "simple";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String simpleValidate(ModelMap map,
        @ModelAttribute("user") UserInfo user, BindingResult result) {

```

```
// Không để trống tên
if (user.getName() == null || user.getName().length() == 0) {
    result.rejectValue("name", null, "Không để trống tên !");
}
// Không để trống tuổi
if (user.getAge() == null) {
    result.rejectValue("age", "required.user.age");
}
// Tuổi phải từ 16 đến 65
else if (user.getAge() < 16 || user.getAge() > 65) {
    result.rejectValue("age", "invalid.user.age");
}
// Nếu có lỗi -> hiển thị view simple
if (result.hasErrors()) {
    return "simple";
}
return "success";
}
```

Với action này chúng ta thấy kiểm tra name và age có để trống hay không đồng thời cũng kiểm tra tuổi có thuộc (16, 65) hay không. Nếu không hợp lệ sẽ gọi phương thức `rejectValue()` của `BindingResult` để bổ sung thông báo lỗi.

Bạn cũng thấy phương thức `rejectValue()` được sử dụng với 2 trường hợp khác nhau:

- ✓ `result.rejectValue("name", null, "Không để trống tên !")`: bổ sung trực tiếp thông báo lỗi cho trường name.
- ✓ `result.rejectValue("age", "required.user.age")`: bổ sung lỗi được định nghĩa trong file tài nguyên (`SimpleValidation.properties`, trình bày ở phần kế tiếp) với mã là `required.user.age`.

Cuối cùng `result.hasErrors()` cho chúng ta biết có lỗi nào trong `result` hay không. Nếu có thì quay lại trang input để hiển thị lỗi yêu cầu nhập lại.

❖ Bước 4: Xây dựng file tài nguyên chứa các thông báo lỗi

File này định nghĩa các thông báo lỗi để được sử dụng trong phần kiểm lỗi. Đây là file từ điển, bạn chỉ cần đưa ra các cặp `key=value` trên mỗi hàng. Key sẽ được sử dụng trong mã java để truy xuất value trong file tài nguyên. File này được đặt trong classpath (trong thư mục `WEB-INF/classes/...`)

```
required.user.age = Không để trống tuổi
invalid.user.age = Tuổi phải từ 16 đến 65
```

❖ Bước 5: Cấu hình

Để Spring biết được file tài nguyên ở đâu mà đọc thì bạn phải khai báo bean `ResourceBundleMessageSource` và chỉ rõ tên danh sách tài nguyên. Nếu trong ứng dụng của bạn có nhiều file tài nguyên thì chỉ việc thêm `<value>TenTaiNguyen</value>`. Chú ý không bao gồm phần mở rộng `properties`.

Spring-config-resources.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <bean id="messageSource"
```

```
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="defaultEncoding" value="utf-8" />
        <property name="basenames">
            <list>
                <value>SimpleValidation</value>
            </list>
        </property>
    </bean>
</beans>
```

Spring-config-mvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- Declare a view resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />
    <!-- Spring MVC Annotation -->
    <mvc:annotation-driven />
    <context:annotation-config />
    <!-- Where to find component -->
    <context:component-scan base-package="com.lycato" />
</beans>
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <display-name>Spring MVC Validation</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config-*.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

❖ Bước 6: Chạy

Chạy ứng dụng như phần mô tả

- ✓ Không nhập
- ✓ Nhập tuổi không hợp lệ
- ✓ Nhập thông tin hợp lệ

5.5.2.2 Kiểm lỗi bằng annotation

Chúng ta thực hiện lại bài kiểm lỗi ở trên nhưng với phương pháp sử dụng annotation kiểm lỗi chuyên dụng.

❖ Bước 1: Hiệu chỉnh UserInfo.java

Giữ nguyên toàn bộ mã UserInfo.java, chỉ đính kèm thêm các annotation kiểm lỗi phù hợp với các trường muốn kiểm lỗi.

- ✓ @NotEmpty: không cho để trống chuỗi name
- ✓ @NotNull: không cho phép age null (để trống)
- ✓ @Range: giới hạn giá trị số

```
@NotEmpty(message="Không để trống tên")
String name;

@NotNull
@Range(min=16, max=65)
Integer age;
```

Tất cả các annotation kiểm lỗi đều có thuộc tính message để thiết lập thông báo lỗi. Tuy nhiên thuộc tính này chỉ được sử dụng khi trong file tài nguyên không định nghĩa thông báo lỗi cho thuộc tính kiểm lỗi đó.

Trong ví dụ này nếu bạn không nhập tên thì chưa chắc dòng thông báo "Không để trống tên" đã được xuất ra mà có khi xuất một thông báo khác nếu trong file tài nguyên có định nghĩa thông báo cho luật này (xem phần file tài nguyên).

❖ Bước 2: File tài nguyên

Khác với kiểm lỗi bằng tay, ở đây key của file tài nguyên phải đặt đúng qui cách. Tên key đầy đủ gồm 3 phần, mỗi phần có { nghĩa khác nhau.

- ✓ Phần 1: tên Annotation kiểm lỗi (NotEmpty)
- ✓ Phần 2: tên command object (user)
- ✓ Phần 3: tên thuộc tính muốn kiểm lỗi (name)

```
NotEmpty.user.name = Không để trống tên

NotNull.user.age = Không để trống tuổi
Range.user.age = Tuổi phải từ 16 đến 65

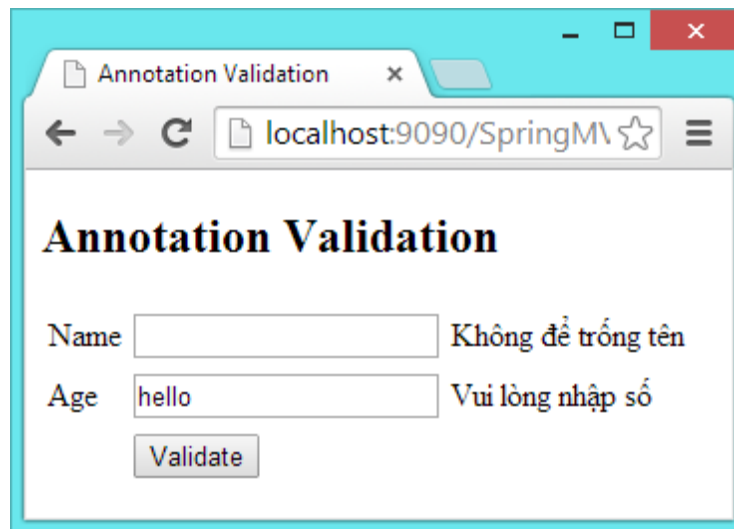
#Sai kiểu dữ liệu
typeMismatch.user.age=Vui lòng nhập số
```

Ví dụ với key NotNull.user.age thì khi thuộc tính age của user bị null thì thông báo "không để trống tuổi" sẽ được truy xuất và hiển thị.

Chú ý:

- ✓ Nếu key là NotNull.user thì tất cả các trường của user bị null thì thông báo trên được hiển thị
- ✓ Nếu key là NotNull thì tất cả các trường của bất kz command object nào cũng được thông báo bởi thông báo này.

Key typeMismatch.user.age định nghĩa thông báo cho thuộc tính age của đối tượng user khi nhập dữ liệu sai kiểu. Với khai báo như trên thì khi bạn nhập dữ liệu không đúng kiểu thì nhận được thông báo "Vui lòng nhập số"



❖ Bước 3: Hiệu chỉnh SimpleController.java

Bổ sung @Valid vào trước UserInfo để Spring MVC tự động kiểm lỗi user theo các annotation đã đính kèm trong UserInfo. Các lỗi (nếu có) sẽ được bổ sung vào result. Nhiệm vụ của chúng ta là kiểm tra xem trong result có lỗi hay không để lựa chọn view hiển thị cho phù hợp. Toàn bộ mã kiểm lỗi viết bằng tay được xóa sạch.

```
@RequestMapping(method = RequestMethod.POST)
public String simpleValidate(ModelMap map, BindingResult result,
    @ModelAttribute("user") @Valid UserInfo user) {
    if(result.hasErrors()){
        return "simple";
    }
    return "success";
}
```

5.5.2.3 Annotation kiểm lỗi

Trong ví dụ ở trên chúng ta chỉ mới làm quen vài annotation. String Spring và Java còn có nhiều annotation khác được sử dụng thường xuyên trong kiểm lỗi.

Nhóm các annotation kiểm lỗi đến từ package javax.validation.constraints là các annotation gốc được Java định nghĩa sẵn. Nhóm này bao gồm các annotation sau:

Annotation	Mô tả	Ví dụ
@DecimalMax(value)	Giá trị decimal tối đa	@DecimalMax("30.00") Double discount;
@DecimalMin(value)	Giá trị decimal tối thiểu	@DecimalMin("5.00") Double discount;

@Digits(integer,fraction)	Số trong phạm vi. Integer=số chữ số tối đa của phần nguyên, fraction=số chữ số tối đa phần thập phân.	@Digits(integer=6, fraction=2) Double price;
@Max(value)	Số nguyên tối đa	@Max(10) Integer quantity;
@Min(value)	Số nguyên tối thiểu	@Min(5) Integer quantity;
@NotNull	Không cho phép null	@NotNull String username;
@Null	Phải là giá trị null	@Null String unusedString;
@Pattern(regex)	Phải khớp với biểu thức chính quy	@Pattern(regex="\d{9}") String idcard;
@Size(min,max)	Phạm vi kích thước của String, Collection, Map, Array	@Size(min=2, max=240) String description;
@Future	Ngày trong tương lai	@Future Date event;
@Past	Ngày trong quá khứ	@Past Date birthday;
@AssertFalse	Chỉ chấp nhận giá trị false	@AssertFalse Boolean supported;
@AssertTrue	Chỉ chấp nhận giá trị true	@AssertTrue Boolean active;
@ScriptAssert	Chỉ chứa mã script	@ScriptAssert String script;

Bên cạnh các annotation gốc, Spring mở rộng để có thêm một số annotation mới thuộc package org.hibernate.validator.constraints. Nhóm này bao gồm:

Annotation	Mô tả	Ví dụ
NotBlank	Chuỗi không rỗng hoặc null	@ NotBlank String name;
Email	Định dạng email	@Email String email;
CreditCardNumber	Định dạng số thẻ tín dụng	@ CreditCardNumber String credicard;
URL	Định dạng url	@URL String home;
Length(min, max)	Giới hạn độ dài chuỗi	@ Length(min=6) String password;
SafeHtml(whitelistType)	NONE, SIMPLE_TEXT, BASIC, BASIC_WITH_IMAGES, RELAXED	@SafeHtml(whitelistType = WhiteListType.NONE) String description;
Range(min, max)	Số trong khoảng	@ Range(min=0, max=10) Double marks;
NotEmpty	string, collection, map hay array không	@NotEmpty String[] hobbies;

	null hoặc rỗng	
--	----------------	--

5.5.2.4 Định nghĩa và sử dụng annotation kiểm lỗi tùy biến

Trong trường hợp danh sách các annotation kiểm lỗi trên vẫn chưa đáp ứng yêu cầu của bạn, Spring cho phép bạn định nghĩa ra các annotation riêng của mình.

Ví dụ sau chúng ta sẽ định nghĩa 2 ràng buộc kiểm lỗi mới là số chẵn EvenNumber và giới hạn tuổi. Mong muốn sẽ được sử dụng như mọi annotation khác:

```
@EvenNumber
Integer namNhuon;

@AgeRange(min=16, max=65)
Date birthday;
```

Để định nghĩa ra một luật kiểm lỗi mới, chúng ta phải tạo ra 2 thành phần:

- ✓ Annotation
- ✓ Lớp kiểm lỗi cho annotation

Để hiểu rõ hơn, chúng ta hãy định nghĩa 2 luật kiểm lỗi đã nói ở trên

- ✓ Luật kiểm số chẵn @EvenNumber

Annotation (EvenNumber.java)

Annotation này phải gắn liền với một lớp kiểm lỗi. Lớp này được chỉ ra bởi @Constraint().

```
@Documented
@Constraint(validatedBy = EvenNumberConstraintValidator.class)
@Target({ ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface EvenNumber {
    String message() default "{EvenNumber} must be an even number";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Lớp kiểm lỗi cho annotation (EvenNumberConstraintValidator)

Theo qui định, lớp này phải thực thi theo interface ConstraintValidator và cài đặt mã cho 2 phương thức qui định là initialize() và isValid().

- ✓ Initialize(): nhận annotation muốn kiểm lỗi
- ✓ isValid(): viết mã kiểm lỗi. Kết quả trả về true là hợp lệ, ngược lại không hợp lệ

```
public class EvenNumberConstraintValidator
    implements ConstraintValidator<EvenNumber, Integer> {
    @Override
    public void initialize(EvenNumber annotation) { }

    @Override
    public boolean isValid(Integer field, ConstraintValidatorContext ctx) {
        if (field == null) {
            return true;
        }
        return field % 2 == 0;
    }
}
```

```
}  
}
```

❖ Luật kiểm giới hạn tuổi @AgeRange(min,max)

Annotation (AgeRange.java)

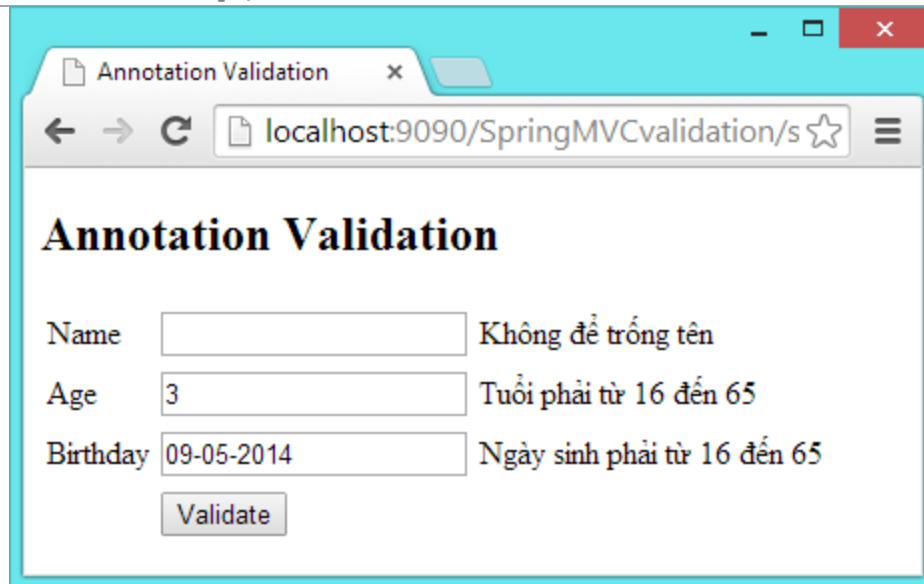
```
@Documented  
@Constraint(validatedBy = AgeRangeNumberConstraintValidator.class)  
@Target({ ElementType.METHOD, ElementType.FIELD })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface AgeRange {  
    int min();  
  
    int max();  
  
    String message() default "The age must be between {min} and {max}";  
  
    Class<?>[] groups() default {};  
  
    Class<? extends Payload>[] payload() default {};  
}
```

Lớp kiểm lỗi cho @AgeRange (AgeRangeConstraintValidator.java)

```
public class AgeRangeNumberConstraintValidator implements  
    ConstraintValidator<AgeRange, Date> {  
    private AgeRange annotation;  
  
    @Override  
    public void initialize(AgeRange annotation) {  
        this.annotation = annotation;  
    }  
  
    @Override  
    public boolean isValid(Date field, ConstraintValidatorContext cxt) {  
        if (field == null) {  
            return true;  
        }  
        Calendar calendar = Calendar.getInstance();  
        calendar.setTime(field);  
        int year = calendar.get(Calendar.YEAR);  
        return year >= annotation.min() && year <= annotation.max();  
    }  
}
```

Sử dụng Annotation tùy biến

Bây giờ chúng ta sẽ sử dụng @AgeRange của mình vừa định nghĩa để kiểm lỗi tuổi đối với ngày sinh. Để tiện chúng ta nâng cấp bài kiểm lỗi ở trên bằng cách thêm vào 1 ô nhập birthday và kiểm lỗi ngày sinh như hình sau:



Bước 1: Bổ sung trường nhập ngày sinh vào simple.jsp

```
<tr>
    <td>Birthday</td>
    <td>
        <form:input path="birthday" />
        <form:errors path="birthday"/>
    </td>
</tr>
```

Bước 2: Bổ sung thuộc tính birthday vào UserInfo.java

```
@DateTimeFormat(pattern="dd-MM-yyyy") /*--Định dạng ngày--*/
@AgeRange(min = 16, max = 65)
Date birthday;

public Date getBirthday() {
    return birthday;
}

public void setBirthday(Date birthday) {
    this.birthday = birthday;
}
```

Bước 3: Bổ sung thông báo lỗi vào file tài nguyên SimpleValidation.properties

```
AgeRange.user.birthday = Ngày sinh phải từ 16 đến 65
```

5.5.2.5 Hiện thị lỗi

Thông báo lỗi được hiện thị trên viên nhờ `<form:errors path="field"/>`. Thông báo lỗi sẽ được hiện thị riêng cho từng trường khác nhau tùy vào thuộc tính path.

Mỗi thông báo lỗi được xuất ra với một thẻ span có id riêng có chứa errors. Ví dụ trường name khi gặp lỗi sẽ xuất thông báo lỗi như sau:

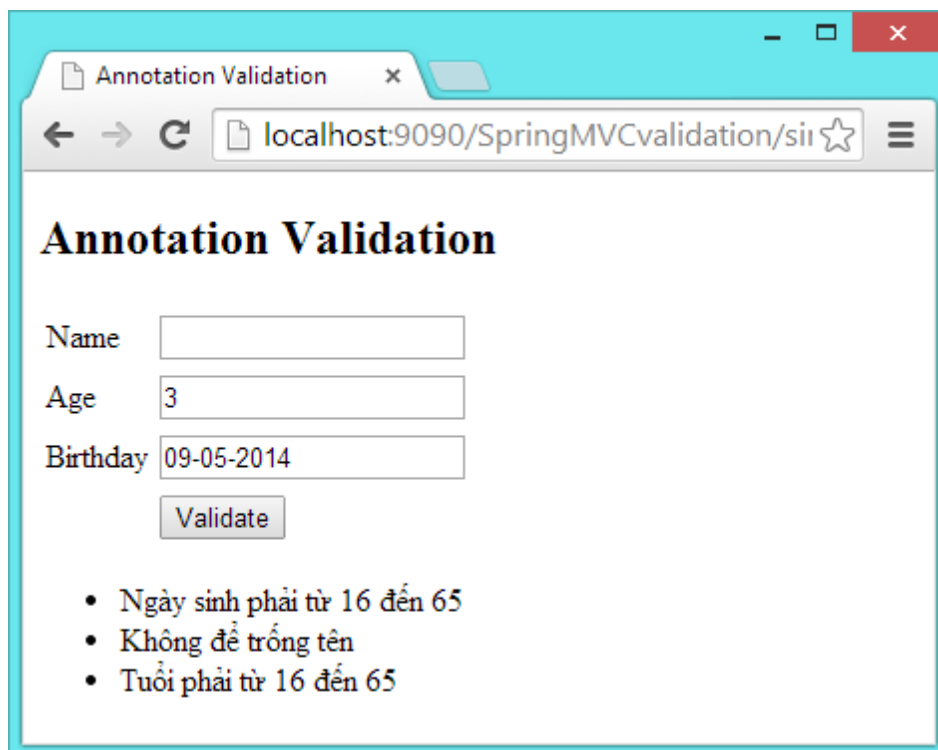
```
<span id="name.errors">Không để trống tên</span>
```

Nếu bạn muốn đổi span thành một thẻ khác (ví dụ ``) thì bạn sử dụng thuộc tính element.

Để hiển thị tất cả các lỗi tại 1 điểm thì thiết lập giá trị cho thuộc tính path là *, nghĩa là `<form:errors path="*/">`. Thông thường các lỗi cách nhau là `
` tức là mỗi thông báo lỗi đặt trên 1 hàng. Nếu bạn muốn thay đổi sự tách biệt này thì điều chỉnh thuộc tính delimiter.

Sau đây là định nghĩa và hiển thị lỗi

```
<ul>
    <form:errors path="*" element="li" delimiter="</li><li>"/>
</ul>
```

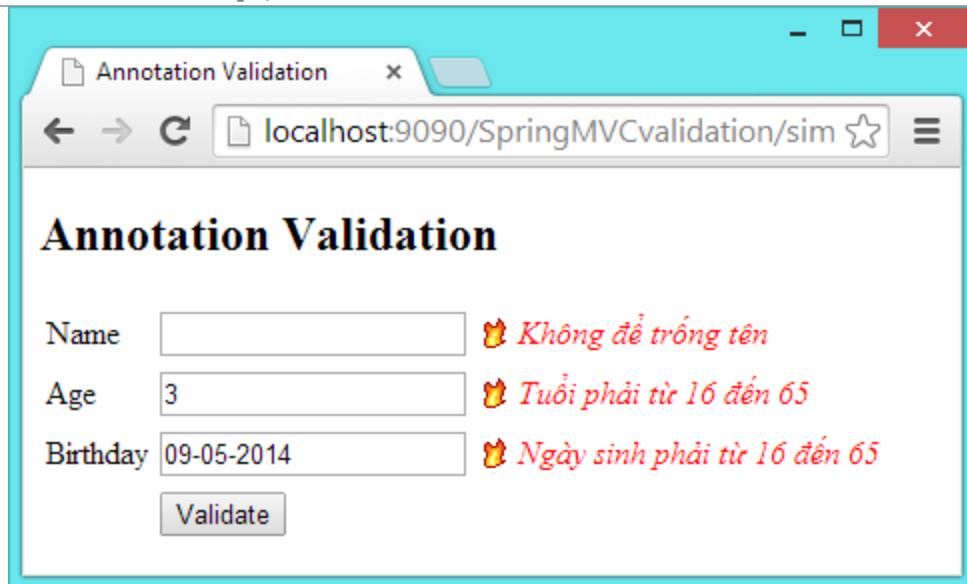


Nếu bạn muốn định dạng thông báo lỗi với css thì chỉ cần định nghĩa css cho các span có id kết thúc với errors là được.

CSS này sẽ cho hiển thị lỗi như hình dưới. Tất nhiên bạn phải có ảnh trong thư mục images.

```
<style type="text/css">
span[id$=errors] {
    color:red;
    font-style: italic;
    padding-left: 20px;
    background: url("images/anifire.gif") no-repeat left center;
}
</style>
```

Như vậy để có được thông báo như sau



5.5.2.6 Thay đổi thông báo mặc định

Thông thường khi phạm một lỗi thì thông báo phải được đưa ra. Khi không có thông báo nào được định nghĩa thì một thông báo mặc định sẽ được sử dụng.

Bạn có thể thay đổi hoàn toàn thông báo mặc định này bằng cách tạo file `ValidationMessages.properties` trong thư mục `src` (hoặc `WEB-INF/classes`) và định nghĩa lại toàn bộ các thông báo lỗi sau đây.

```

javax.validation.constraints.AssertFalse.message = must be false
javax.validation.constraints.AssertTrue.message = must be true
javax.validation.constraints.DecimalMax.message = must be less than or equal to {value}
javax.validation.constraints.DecimalMin.message = must be greater than or equal to {value}
javax.validation.constraints.Digits.message = numeric value out of bounds (<{integer}
digits>.{fraction} digits> expected)
javax.validation.constraints.Future.message = must be in the future
javax.validation.constraints.Max.message = must be less than or equal to {value}
javax.validation.constraints.Min.message = must be greater than or equal to {value}
javax.validation.constraints.NotNull.message = may not be null
javax.validation.constraints.Null.message = must be null
javax.validation.constraints.Past.message = must be in the past
javax.validation.constraints.Pattern.message = must match "{regexp}"
javax.validation.constraints.Size.message = size must be between {min} and {max}

org.hibernate.validator.constraints.CreditCardNumber.message = invalid credit card number
org.hibernate.validator.constraints.Email.message = not a well-formed email address
org.hibernate.validator.constraints.Length.message = length must be between {min} and {max}
org.hibernate.validator.constraints.NotBlank.message = may not be empty
org.hibernate.validator.constraints.NotEmpty.message = may not be empty
org.hibernate.validator.constraints.Range.message = must be between {min} and {max}
org.hibernate.validator.constraints.SafeHtml.message = may have unsafe html content
org.hibernate.validator.constraints.ScriptAssert.message = script expression "{script}" didn't evaluate
to true
org.hibernate.validator.constraints.URL.message = must be a valid URL

typeMismatch=type mismatch
    
```

5.6 LÀM VIỆC VỚI CSDL

Bên cạnh JDBC là lý thuyết chung để làm việc với CSDL quan hệ Spring còn cung cấp JdbcTemplate để đơn giản hóa công việc này nhưng mang lại hiệu quả tốt hơn bội phần.

Không những thế Spring còn cho phép chúng ta tích hợp một cách dễ dàng với Hibernate – một framework làm việc với CSDL nổi tiếng nhất hiện nay.

5.6.1 JdbcTemplate

5.6.1.1 Cấu hình bean JdbcTemplate

Để sử dụng JdbcTemplate trong Spring MVC, bạn cần phải cấu hình 2 bean:

- ✓ DataSource: bean này giúp cấu hình kết nối đến CSDL làm việc. Tùy thuộc vào CSDL mà bạn cần phải nạp driver tương ứng để làm việc và cấu hình phù hợp.
- ✓ JdbcTemplate: bean này giúp bạn sử dụng JdbcTemplate API để thực hiện truy vấn và thao tác dữ liệu.

spring-config-jdbc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource"
        p:driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"
        p:url="jdbc:sqlserver://localhost:1433;DatabaseName=EshopV10"
        p:username="sa" p:password="*****" />
    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dataSource" />
    </bean>
</beans>
```

- ✓ Bean dataSource giúp khai báo kết nối đến CSDL. Trong trường hợp này SQL Server vì vậy bạn cần nạp thư viện sqljdbc4.jar chứa driver cho phép làm việc với CSDL SQL Server. Hãy chú ý đến các thông số kết nối như driverClassName, url, username, password.
- ✓ Bean jdbcTemplate chứa API mà bạn sẽ tiêm vào ứng dụng để sử dụng. Bean này cần liên kết với bean dataSource để biết CSDL muốn làm việc.

5.6.1.2 JdbcTemplate API

Với JdbcTemplate bạn có thể thực hiện mọi thao tác cập nhật cũng như truy vấn một cách ngắn gọn. Sau đây là một số phương thức thường được sử dụng của JdbcTemplate.

int update(String sql, Object...args)

- ✓ Công dụng: Thực thi câu lệnh sql INSERT, UPDATE và DELETE.
- ✓ Tham số:
 - Sql: câu lệnh cập nhật dữ liệu

- Object...args: mảng chứa các giá trị cung cấp cho các tham số
- ✓ Kết quả: Số bản ghi cập nhật được

Ví dụ:

```
String sql = "UPDATE Categories SET Name=?, NameVN=? WHERE Id=?";
jdbcTemplate.update(sql, "Mobile", "Điện thoại di động", 1005)
```

List<T> query(String sql, RowMapper<T> rowMap, Object...args)

- ✓ Công dụng: Thực thi câu lệnh truy vấn sql SELECT.
- ✓ Tham số:
 - Sql: câu lệnh truy vấn dữ liệu
 - RowMapper<T> rowMap: Kiểu chứa một bản ghi dữ liệu
 - Object...args: mảng chứa các giá trị cung cấp cho các tham số
- ✓ Kết quả: Danh sách các đối tượng chứa bản ghi dữ liệu

Ví dụ:

```
String sql = "SELECT * FROM Categories WHERE Name LIKE ?";
RowMapper<Category> rowMapper=new BeanPropertyRowMapper<Category>(Category.class)
List<Category> rows = jdbcTemplate.query(sql, rowMapper , "%on%")
```

<T> queryForObject(String sql, RowMapper<T> rowMap, Object...args)

- ✓ Công dụng: Thực thi câu lệnh truy vấn sql SELECT trả về một bản ghi
- ✓ Tham số:
 - Sql: câu lệnh truy vấn dữ liệu
 - RowMapper<T> rowMap: Kiểu chứa một bản ghi dữ liệu.
 - Object...args: mảng chứa các giá trị cung cấp cho các tham số
- ✓ Kết quả: Đối tượng chứa một bản ghi dữ liệu đọc được

Ví dụ:

```
String sql = "SELECT * FROM Categories WHERE id=?";
RowMapper<Category> rowMapper=new BeanPropertyRowMapper<Category>(Category.class)
Student row = jdbcTemplate.queryForObject(sql, rowMapper, 1005 )
```

XYZ queryForXYZ(String sql, Object...args)

- ✓ Công dụng: Thực thi câu lệnh truy vấn sql SELECT trả về một giá trị
- ✓ Tham số:
 - Sql: câu lệnh truy vấn dữ liệu
 - Object...args: mảng chứa các giá trị cung cấp cho các tham số
- ✓ Kết quả: Giá trị truy vấn được

Ví dụ:

```
String sql = "SELECT COUNT(*) FROM Categories";  
int rows = jdbcTemplate.queryForInt(sql)
```

5.6.1.3 Thao tác dữ liệu với JdbcTemplate

Để thực hiện các thao tác dữ liệu như INSERT, UPDATE và DELETE với JdbcTemplate là rất đơn giản. Bạn chỉ cần thực hiện 3 bước:

Bước 1: Tiêm jdbcTemplate vào Controller

- ✓ @Autowired JdbcTemplate jdbc

Bước 2: Xây dựng câu lệnh sql

- ✓ sql = "INSERT INTO Categories(Name, NameVN) VALUES(?, ?)";
- ✓ sql = "UPDATE Categories SET Name=?, NameVN=? WHERE Id=?";
- ✓ sql = "DELETE FROM Categories WHERE Id=?";

Bước 3: gọi phương thức update(sql) để thực thi câu lệnh

- ✓ jdbc.update(sql)

Ví dụ: Action /category/insert tiếp nhận thông tin Category sau đó thực hiện câu lệnh insert để chèn dữ liệu vào bảng Categories.

```
@Controller  
public class JdbcController {  
    @Autowired  
    JdbcTemplate jdbc; // tiêm bean jdbcTemplate  
  
    @RequestMapping("/category/insert")  
    public String insertCategory(ModelMap map,  
        @ModelAttribute("category") Category cate) {  
        String sql = "INSERT INTO Categories(Name, NameVN) VALUES(?, ?)";  
        // thực thi câu lệnh thao tác dữ liệu jdbc.update(sql, cate.getName(),  
        // cate.getNamevn());  
        return "category";  
    }  
}
```

Trong đó Category

```
public class Category implements Serializable{  
    Integer id;  
    String name;  
    String namevn;  
  
    Getters/setters  
}
```

5.6.1.4 Truy vấn dữ liệu với JdbcTemplate

Trong các phương thức truy vấn dữ liệu của JdbcTemplate được giới thiệu ở trên thì query() và queryForObject() có đối số là RowMapper, nó qui định cách đọc dữ liệu của mỗi bản ghi.

Trong các ví dụ này chúng tôi sử dụng BeanPropertyRowMapper (đã được JdbcTemplate cung cấp sẵn) để đọc dữ liệu dựa vào các thuộc tính của bean.

5.6.1.4.1 BeanPropertyRowMapper

Để thực hiện truy vấn dữ liệu từ bảng Categories bạn chỉ cần thực hiện theo các bước hướng dẫn sau

Bước 1: Tiêm JdbcTemplate vào Controller

- ✓ @Autowired JdbcTemplate jdbc

Bước 2: Xây dựng câu lệnh truy vấn dữ liệu

- ✓ String sql = "SELECT * FROM Categories";
- ✓ String sql = "SELECT * FROM Categories WHERE Id=?";

Bước 3: Thực hiện truy vấn

- ✓ List<Category> many = jdbc.query(sql, new BeanPropertyRowMapper<Category>())
- ✓ Category one = jdbc.query(sql, new BeanPropertyRowMapper<Category>(), 1005)

Bước 4: Duyệt và xử lý dữ liệu truy vấn được

```
for(Category item : many){  
    // Xử lý item  
}
```

Ví dụ: Sử dụng BeanPropertyRowMapper để truy vấn dữ liệu từ bảng Categories.

```
@Controller  
public class JdbcController {  
    @Autowired  
    JdbcTemplate jdbc; // tiêm bean jdbcTemplate  
  
    @RequestMapping("/category/list")  
    public String listCategory(ModelMap map,  
        @ParamRequest("search") String search) {  
        String sql = "SELECT * FROM Categories WHERE Name LIKE ?";  
        List<Category> list = jdbc.query(sql,  
            new BeanPropertyRowMapper<Category>(), search);  
        map.addAttribute("categories", list);  
        return "category";  
    }  
}
```

5.6.1.4.2 RowMapper

Interface RowMapper giúp chúng ta xây dựng lớp đọc dữ liệu một bản ghi chuyển đổi thành đối tượng chứa dữ liệu của mỗi bản ghi.

Để đọc dữ liệu mỗi bản ghi của bảng Categories chúng ta xây dựng lớp CategoryRowMapper thực thi theo interface RowMapper và viết mã thực hiện đọc dữ liệu cho mỗi bản ghi như sau:

```
public class CategoryRowMapper implements RowMapper<Category> {  
    @Override  
    public Category mapRow(ResultSet rs, int index) throws SQLException {
```

```

        Category cate = new Category();
        cate.setId(rs.getInt("Id"));
        cate.setName(rs.getString("Name"));
        cate.setNamevn(rs.getString("NameVN"));
        return cate;
    }
}

```

Như bạn thấy, interface này qui định phải viết mã cho phương thức `mapRow(ResultSet, int)`. Phương thức này được gọi khi `JdbcTemplate` duyệt qua từng bản ghi. Nhiệm vụ của chúng ta là đọc dữ liệu từ `ResultSet` và tạo ra đối tượng dữ liệu và nó trả về theo qui định của phương thức.

Sau khi bạn đã viết lớp này, bạn có thể sử dụng trong các phương thức `query()`, `queryForObject` để truy vấn dữ liệu.

- ✓ `List<Category> many = jdbc.query(sql, new CategoryRowMapper())`
- ✓ `Category one = jdbc.queryForObject(sql, new CategoryRowMapper(), 1005)`

Ví dụ: Sử dụng `CategoryRowMapper` để truy vấn dữ liệu từ bảng `Categories`.

```

@Controller
public class JdbcController {
    @Autowired
    JdbcTemplate jdbc; // tiêm bean jdbcTemplate

    @RequestMapping("/category/list")
    public String listCategory(ModelMap map,
        @ParamRequest("search") String search) {
        String sql = "SELECT * FROM Categories WHERE Name LIKE ?";
        List<Category> list = jdbc.query(sql, new CategoryRowMapper(), search);
        map.addAttribute("categories", list);
        return "category";
    }
}

```

5.6.2 Tích hợp Hibernate

5.6.2.1 Khai báo cấu hình tích hợp

Hiệu chỉnh các thông số kết nối đến CSDL phù hợp đồng thời khai báo package chứa các entity class trong file cấu hình tích hợp Hibernate sau đây

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <!-- DataSource -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"
        p:url="jdbc:sqlserver://localhost; Database=EShopV10"
        p:username="sa"
        p:password="songlong"/>

    <!-- Hibernate Session Factory -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

```



```

<property name="dataSource" ref="dataSource"/>
<property name="hibernateProperties">
  <props>
    <prop key="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</prop>
    <prop key="hibernate.show_sql">>false</prop>
  </props>
</property>
<property name="packagesToScan" value="eshop.entity"/>
</bean>

<!-- Transaction Manager -->
<bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory" />
<tx:annotation-driven transaction-manager="transactionManager" />

</beans>

```

5.6.2.2 Lập trình Hibernate trong Controller

Khai khi đã khai báo cấu hình tích hợp, trong Controller chỉ cần thêm SessionFactory vào là bạn có thể lập trình được với Hibernate.

```

@Controller
@RequestMapping(value="account")
public class AccountController {
    @Autowired
    SessionFactory factory;

    @RequestMapping(value="register", method=RequestMethod.POST)
    public String register(ModelMap model,
        @ModelAttribute("customer") Customer customer) {
        Session session = factory.openSession();
        Transaction transaction = session.beginTransaction();
        try {
            session.save(customer);
            transaction.commit();
            model.addAttribute("message", "Đăng ký thành công !");
        }
        catch (Exception e) {
            transaction.rollback();
            model.addAttribute("message", "Đăng ký thất bại !");
            e.printStackTrace();
        }
        session.close();
        return "register";
    }
}

```

5.6.2.3 Điều khiển transaction

Ở trên bạn mở một session mới và viết mã điều khiển transaction. Khi tích hợp Hibernate bạn có thể giao việc điều khiển transaction cho Spring bằng cách sử dụng @Transactional đặt lên action nào bạn muốn hoặc đặt trên class nếu bạn muốn tất cả các action đều được Spring điều khiển transaction.

Khi sử dụng @Transactional thì một session được mở sẵn để dùng chung. Để tham chiếu đến session này bạn sử dụng session.getCurrentSession() thay vì factory.openSession() để mở một session mới.

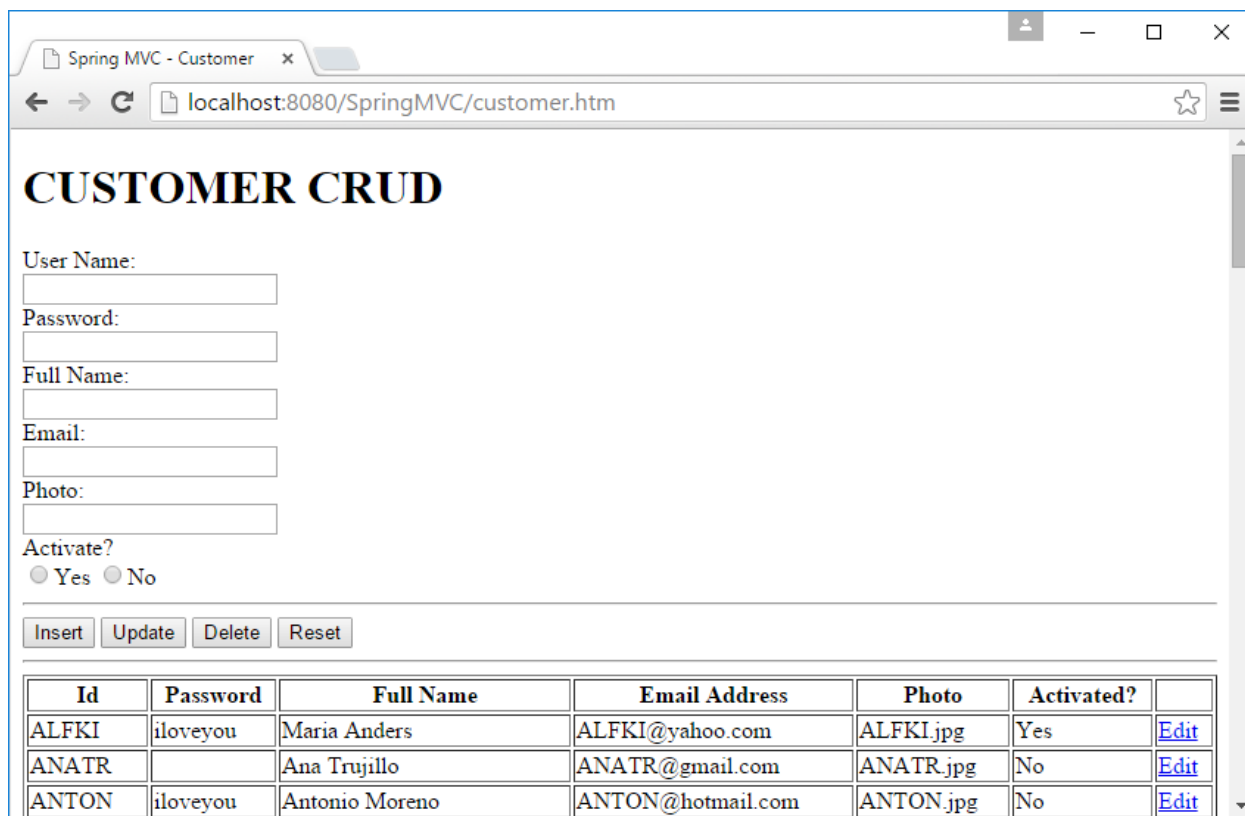
Tất nhiên bạn vẫn có thể mở một session mới để tự điều khiển các transaction riêng của mình.

```
@Controller
@Transactional
@RequestMapping(value="account")
public class AccountController {
    @Autowired
    SessionFactory factory;

    @RequestMapping(value="login", method=RequestMethod.POST)
    public String login(ModelMap model,
        @RequestParam("id") String id,
        @RequestParam("password") String pw) {
        Session session = factory.getCurrentSession();
        ...
        return "login";
    }

    @RequestMapping(value="register", method=RequestMethod.GET)
    public String editProfile(ModelMap model, @RequestParam("id") String id) {
        Session session = factory.getCurrentSession();
        ...
        return "register";
    }

    @RequestMapping(value="register", method=RequestMethod.POST)
    public String register(ModelMap model,
        @ModelAttribute("customer") Customer customer) {
        Session session = factory.openSession();
        Transaction transaction = session.beginTransaction();
        try {
            session.save(customer);
            transaction.commit();
            model.addAttribute("message", "Đăng ký thành công !");
        }
        catch (Exception e) {
            transaction.rollback();
            model.addAttribute("message", "Đăng ký thất bại !");
            e.printStackTrace();
        }
        session.close();
        return "register";
    }
}
```

5.6.2.4 Ví dụ về Customer CRUD


CUSTOMER CRUD

User Name:

Password:

Full Name:

Email:

Photo:

Activate?
☐ Yes ☐ No

Id	Password	Full Name	Email Address	Photo	Activated?	
ALFKI	iloveyou	Maria Anders	ALFKI@yahoo.com	ALFKI.jpg	Yes	Edit
ANATR		Ana Trujillo	ANATR@gmail.com	ANATR.jpg	No	Edit
ANTON	iloveyou	Antonio Moreno	ANTON@hotmail.com	ANTON.jpg	No	Edit

Để thực hiện quản lý khách hàng, cần xây dựng các thành phần sau:

- CustomerController.java
- Customer.jsp

CustomerController.java

```
package nhatnghe.controller;

...

@Controller
@RequestMapping("customer")
@Transactional
public class CustomerController {
    @Autowired
    SessionFactory factory;

    @ModelAttribute("users")
    public List<Customer> getCustomers(){
        Session session = factory.getCurrentSession();
        String hql = "FROM Customer";
        Query query = session.createQuery(hql);
        List<Customer> list = query.list();
        return list;
    }

    @RequestMapping()
    public String index(ModelMap model) {
        model.addAttribute("user", new Customer());

        return "customer";
    }
}
```

```
}

@RequestMapping(params="btnInsert")
public String insert(ModelMap model,
    @ModelAttribute("user") Customer customer) {

    Session session = factory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        session.save(customer);
        transaction.commit();
        model.addAttribute("message", "Insert successfully !");
    }
    catch (Exception e) {
        model.addAttribute("message", "Insert fails !");
        transaction.rollback();
    }
    session.close();

    model.addAttribute("user", new Customer());
    model.addAttribute("users", getCustomers());

    return "customer";
}

@RequestMapping(params="btnUpdate")
public String update(ModelMap model,
    @ModelAttribute("user") Customer customer) {

    Session session = factory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        session.update(customer);
        transaction.commit();
        model.addAttribute("message", "Update successfully !");
    }
    catch (Exception e) {
        model.addAttribute("message", "Update fails !");
        transaction.rollback();
    }
    session.close();

    model.addAttribute("users", getCustomers());
    return "customer";
}

@RequestMapping(params="btnDelete")
public String delete(ModelMap model, Customer customer) {

    Session session = factory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        session.delete(customer);
        transaction.commit();
        model.addAttribute("message", "Delete successfully !");
    }
    catch (Exception e) {
        model.addAttribute("message", "Delete fails !");
        transaction.rollback();
    }
    session.close();
}
```

```

        model.addAttribute("user", new Customer());
        model.addAttribute("users", getCustomers());
        return "customer";
    }

    @RequestMapping(params="lnkEdit")
    public String edit(ModelMap model, @RequestParam("id") String id) {
        Session session = factory.getCurrentSession();
        Customer user = (Customer) session.get(Customer.class, id);

        model.addAttribute("user", user);
        return "customer";
    }
}

```

Customer.jsp

```

<%@ page pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC - Customer</title>
</head>
<body>
    <h1>CUSTOMER CRUD</h1>
    <h4>${message}</h4>
    <form:form action="customer.htm" modelAttribute="user">
        <div>User Name:</div>
        <form:input path="id"/>

        <div>Password:</div>
        <form:password path="password" showPassword="true"/>

        <div>Full Name:</div>
        <form:input path="fullname"/>

        <div>Email:</div>
        <form:input path="email"/>

        <div>Photo:</div>
        <form:input path="photo"/>

        <div>Activate?</div>
        <form:radiobutton path="activated" value="true"/>Yes
        <form:radiobutton path="activated" value="false"/>No

        <hr>
        <button name="btnInsert">Insert</button>
        <button name="btnUpdate">Update</button>
        <button name="btnDelete">Delete</button>
        <button>Reset</button>
    </form:form>
    <hr>
    <table border="1" style="width:100%">
    <tr>
        <th>Id</th>

```

```

        <th>Password</th>
        <th>Full Name</th>
        <th>Email Address</th>
        <th>Photo</th>
        <th>Activated?</th>
        <th></th>
    </tr>
    <c:forEach var="u" items="{users}">
    <tr>
        <td>${u.id}</td>
        <td>${u.password}</td>
        <td>${u.fullname}</td>
        <td>${u.email}</td>
        <td>${u.photo}</td>
        <td>${u.activated?'Yes':'No'}</td>
        <td><a href="customer.htm?lnkEdit&id=${u.id}">Edit</a></td>
    </tr>
    </c:forEach>
</table>
</body>
</html>

```

5.7 Ajax

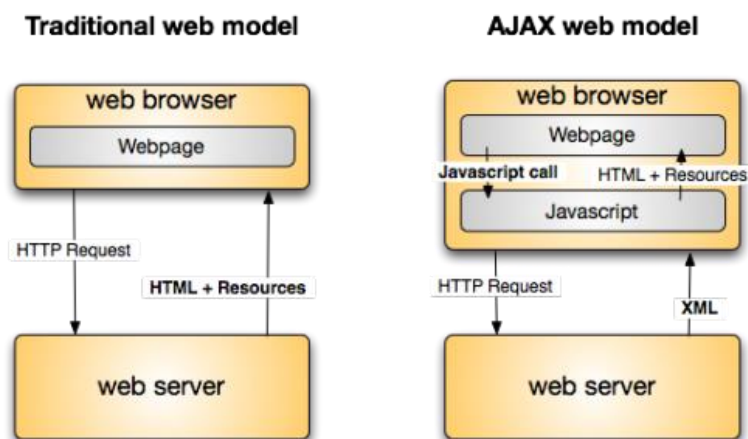
AJAX (Asynchronous Javascript and XML) là kỹ thuật lập trình sử dụng javascript truyền thông bất đồng bộ với server.

Dữ liệu nhận được từ server có thể là Text, HTML, XML, JSON. Trong đó JSON được sử dụng phổ biến hiện nay bởi dữ liệu truyền là text, đối tượng hay mảng đối tượng javascript nên rất nhẹ.

Đặc điểm của kỹ thuật lập trình này

- ✓ Tương tác nhanh với server (không tải lại toàn trang)
- ✓ Dễ học (sử dụng công nghệ sẵn có là javascript, HTML, Text...)
- ✓ Được hỗ trợ nhiều trình duyệt
- ✓ Có nhiều framework mã nguồn mở như jquery, angula...
- ✓ Cộng đồng sử dụng rất lớn -> dễ tìm tài liệu và hỗ trợ

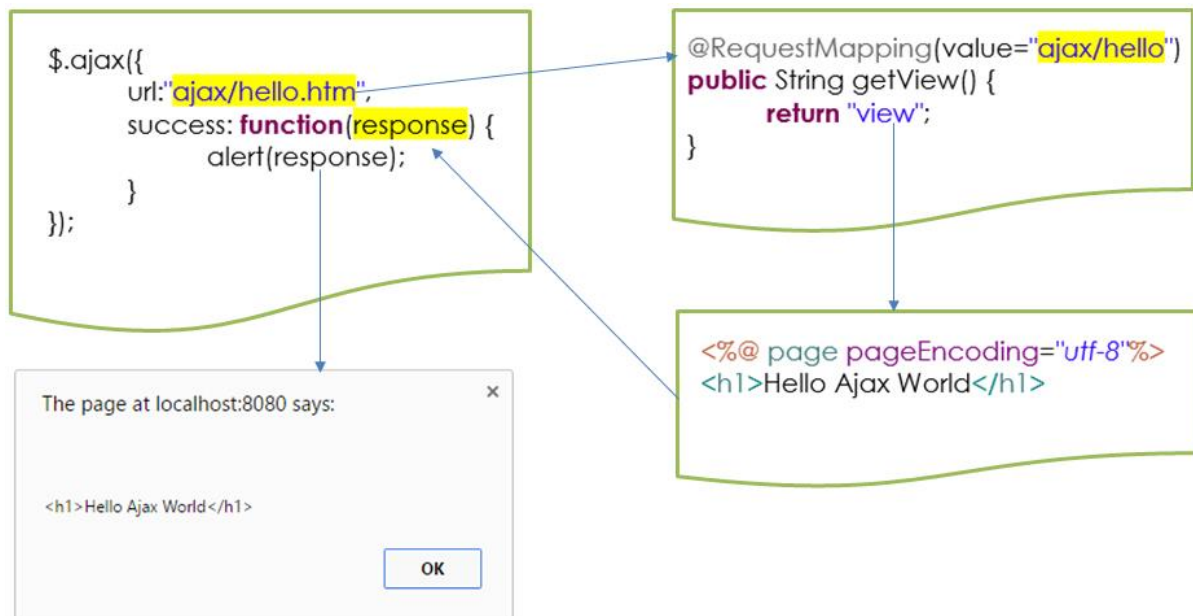
Để hiểu được nguyên lý hoạt động của ajax, chúng ta nghiên cứu 2 mô hình tổ chức sau



Theo mô hình truyền thống: cứ mỗi tương tác của người dùng sẽ sinh yêu cầu gửi đến server. Trong lúc yêu cầu được gửi đi thì người dùng phải đợi cho đến khi kết quả được trả về từ server thì mới có thể thực hiện thao tác tiếp theo.

Theo mô hình ajax: yêu cầu sẽ được javascript gửi ngầm đến server chứ không phải trình duyệt. Nhờ vậy mà trong khi yêu cầu gửi đi để xử lý, người dùng có thể thực hiện những tương tác khác mà không phải chờ đợi kết quả từ phía server. Khi kết quả được trả về thì javascript có trách nhiệm cập nhật những phần giao diện có thay đổi bởi yêu cầu được gửi trước đó.

Để rõ hơn chúng ta xét một ví dụ đơn giản sau đây



Ở ví dụ này chúng ta thấy action ajax/hello.htm được yêu cầu bởi ajax. Phương thức getView() được thực hiện và chọn view.jsp để kết xuất kết quả. Client sẽ nhận được kết quả và sử dụng alert() để thông báo. Quá trình này xảy ra mà trang web không bị refresh.

Để nghiên cứu kỹ thuật lập trình này một cách hiệu quả chúng ta xem xét cả 2 phía: server và client.

5.7.1 Lập trình phía server

Phía server bạn có thể sử dụng php, asp, servlet, jsp... Với chúng ta sử dụng Spring MVC cho phần lập trình phía server.

Với kỹ thuật ajax client chỉ quan tâm đến loại dữ liệu trả về từ server mà không quan tâm đến server sẽ làm công việc gì. Trong phần này chúng ta nghiên cứu các loại dữ liệu được trả về từ server gồm

- ✓ Văn bản hoặc HTML
- ✓ Đối tượng JSON
- ✓ Mảng đối tượng JSON

5.7.1.1 Văn bản hoặc HTML

5.7.1.1.1 Trả về HTML

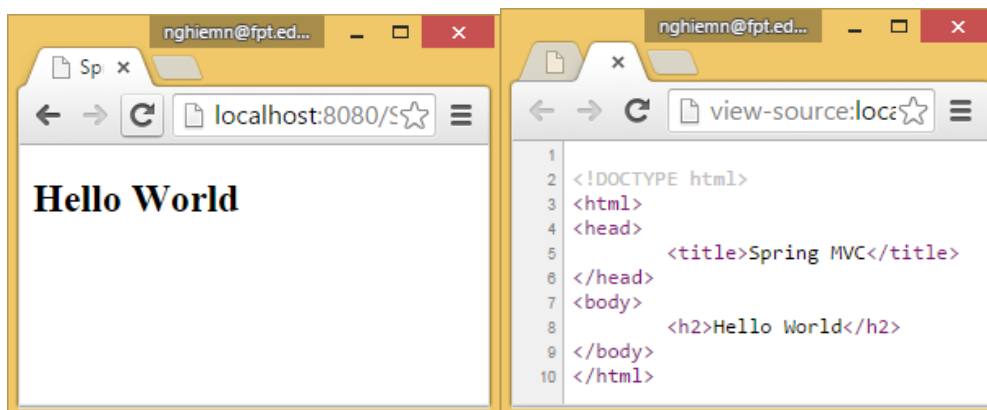
Đây là trường hợp chúng ta đã thực hiện từ trước đến nay – kết quả được trả về là kết xuất từ một view. Giả sử ta có request đến action sau đây

```
@RequestMapping("get-html")
public String getHtml(ModelMap model) {
    model.addAttribute("message", "Hello World");
    return "html";
}
```

Và view html.jsp có nội dung như sau

```
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Spring MVC</title>
</head>
<body>
    <h2>${message}</h2>
</body>
</html>
```

Kết quả nhận được sẽ là kết xuất của view html.jsp

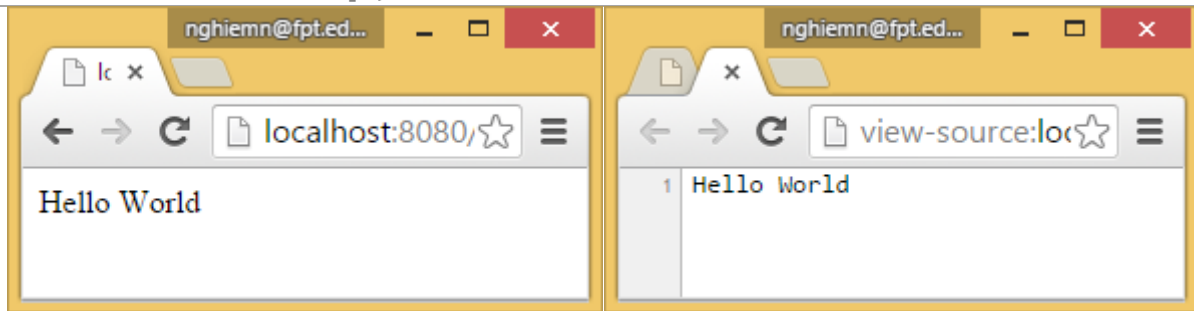


5.7.1.1.2 Trả về văn bản thuần

Để trả về văn bản thuần, action được request được đính với `@ResponseBody`. Khi đó dữ liệu được trả về là chuỗi từ lệnh `return` của action mà không phải là kết quả kết xuất từ view như thường lệ.

```
@ResponseBody
@RequestMapping("get-text")
public String getText() {
    return "Hello World";
}
```

Khi bạn có request đến action này thì sẽ nhận được là



Rõ ràng client nhận được đúng kết quả trả về từ lệnh return mà không phải kết xuất từ một view Hello World.jsp.

5.7.1.2 Đối tượng JSON

Để kết xuất đối tượng JSON chúng ta có thể xuất trực tiếp chuỗi mô tả đối tượng hoặc sử dụng ObjectMapper để chuyển đổi Map hoặc bean sang đối tượng JSON.

Cả 3 action sau đây khi được yêu cầu đều trả về kết quả là đối tượng JSON gồm 2 thuộc tính là count và amount {count:5, amount:100}

```
@ResponseBody
@RequestMapping("get-json")
public String getJson() {
    return "{\"count\":5, \"amount\":100}";
}

@ResponseBody
@RequestMapping("get-map")
public String getMap() {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("count", 5);
    map.put("amount", 100);

    try {
        ObjectMapper mapper = new ObjectMapper();
        String text = mapper.writeValueAsString(map);

        return text;
    }
    catch (Exception e) {
        return "Lỗi";
    }
}

@ResponseBody
@RequestMapping("get-object")
public String getObject() {
    CartInfo cart = new CartInfo();
    cart.setCount(5);
    cart.setAmount(100);

    try {
        ObjectMapper mapper = new ObjectMapper();
        String text = mapper.writeValueAsString(cart);

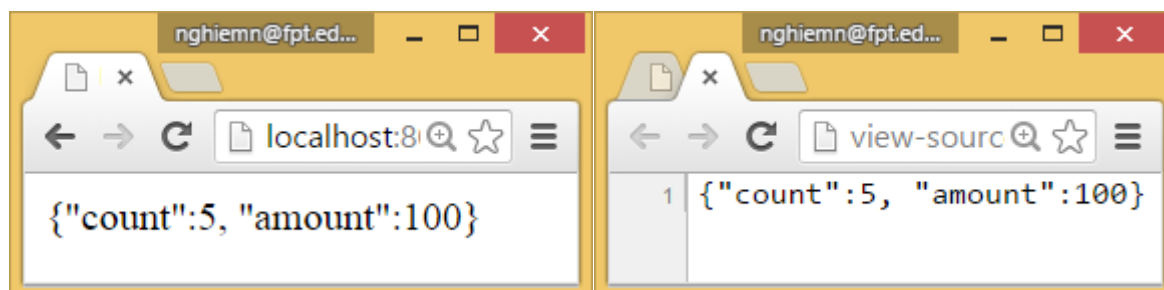
        return text;
    }
    catch (Exception e) {
```

```
        return "Lỗi";
    }
}
```

Trong đó CartInfo có mã như sau

```
public class CartInfo {
    private int count;
    private double amount;
    public int getCount() {
        return count;
    }
    public void setCount(int count) {
        this.count = count;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

Kết quả truy khi truy cập các action trên



5.7.1.3 Mảng đối tượng JSON

Mảng đối tượng JSON có thể được sinh ra từ List<Object> hoặc Object[] sử dụng ObjectMapper.

Cả 2 action sau điều xuất mảng gồm 2 đối tượng JSON [{"count":5, "amount":100}, {"count":10, "amount":200}]

```
@ResponseBody
@RequestMapping("get-list")
public String getList() {
    CartInfo cart1 = new CartInfo();
    cart1.setCount(5);
    cart1.setAmount(100);

    CartInfo cart2 = new CartInfo();
    cart2.setCount(10);
    cart2.setAmount(200);

    List<CartInfo> list = new ArrayList<CartInfo>();
    list.add(cart1);
    list.add(cart2);
}
```

```

    try {
        ObjectMapper mapper = new ObjectMapper();
        String text = mapper.writeValueAsString(list);

        return text;
    }
    catch (Exception e) {
        return "Lỗi";
    }
}

@ResponseBody
@RequestMapping("get-array")
public String getArray() {
    CartInfo[] array = new CartInfo[2];

    array[0] = new CartInfo();
    array[0].setCount(5);
    array[0].setAmount(100);

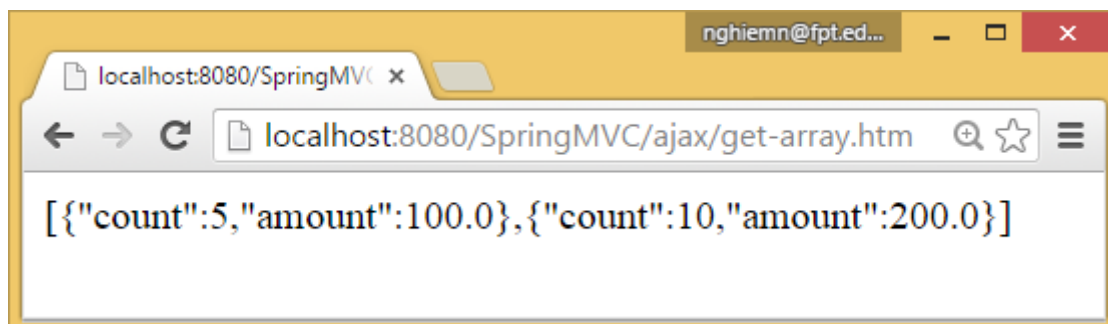
    array[1] = new CartInfo();
    array[1].setCount(10);
    array[1].setAmount(200);

    try {
        ObjectMapper mapper = new ObjectMapper();
        String text = mapper.writeValueAsString(array);

        return text;
    }
    catch (Exception e) {
        return "Lỗi";
    }
}

```

Kết quả truy cập 2 action trên



5.7.2 Lập trình phía client

`$.ajax({})` là hàm được cung cấp bởi jquery để tương tác ajax với server. Hàm này nhận một đối tượng gồm các thuộc tính thường dùng là url, success, dataType, type... Sau đây là cú pháp

```

$.ajax({
    url: <địa chỉ trang web cần tương tác>,
    data:{<danh sách tham số gửi đến server>},

```

type: <phương thức web yêu cầu>,

dataType: <loại dữ liệu nhận được từ server>

success: function(response){ <xử lý kết quả trả về> }

});

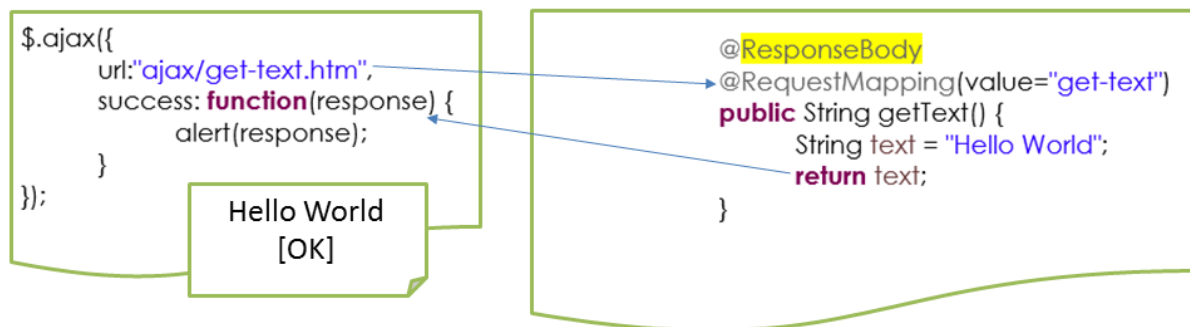
Phía client chỉ quan tâm đến loại dữ liệu trả về từ server mà không quan tâm đến kỹ thuật lập trình phía server. Nói tóm lại, client chỉ quan tâm đến 3 loại dữ liệu sau:

- ✓ Text/HTML
- ✓ Đối tượng JSON
- ✓ Mảng đối tượng JSON

Mỗi loại dữ liệu sẽ được xử lý theo một cách khác nhau.

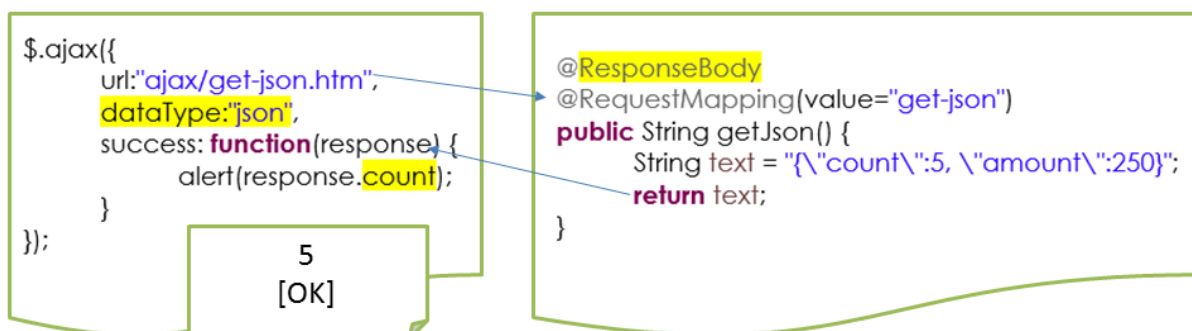
5.7.2.1 Xử lý Text/HTML

Yêu cầu ajax có thể gửi đến các action get-text.htm hoặc get-HTML.htm để nhận được text hoặc html. Kết quả nhận được sẽ được thông báo bằng alert()



5.7.2.2 Xử lý đối tượng JSON

Yêu cầu ajax có thể gửi đến các action get-json.htm, get-map.htm hoặc get-object.htm để nhận được đối tượng json. Chú ý thuộc tính dataType phải có giá trị là json. Kết quả nhận được sẽ là một đối tượng gồm 2 thuộc tính count và amount. alert() được sử dụng để thông báo giá trị của thuộc tính count.



5.7.2.3 Xử lý mảng đối tượng JSON

Yêu cầu ajax có thể gửi đến các action get-list.htm hoặc get-array.htm để nhận được mảng đối tượng json. Thuộc tính dataType phải có giá trị là json. Kết quả nhận được sẽ là mảng

gồm 2 đối tượng json, mỗi đối tượng gồm 2 thuộc tính count và amount. alert() được sử dụng để thông báo giá trị của thuộc tính count của mỗi đối tượng.

```
$.ajax({
  url:"ajax/get-list.htm",
  dataType:"json",
  success: function(response) {
    $(response).each(function(index, item){
      alert(index + "=" + item.count)
    });
  }
});
```

```
@ResponseBody
@RequestMapping("get-list")
public String getList() {
  CartInfo cart1 = new CartInfo();
  cart1.setCount(5);
  cart1.setAmount(100);
  CartInfo cart2 = new CartInfo();
  cart2.setCount(10);
  cart2.setAmount(200);
  List<CartInfo> list = new ArrayList<CartInfo>();
  list.add(cart1);
  list.add(cart2);
  try {
    ObjectMapper mapper = new ObjectMapper();
    String text = mapper.writeValueAsString(list);
    return text;
  } catch (Exception e) {
    return "Lỗi";
  }
}
```

0=5
[OK]

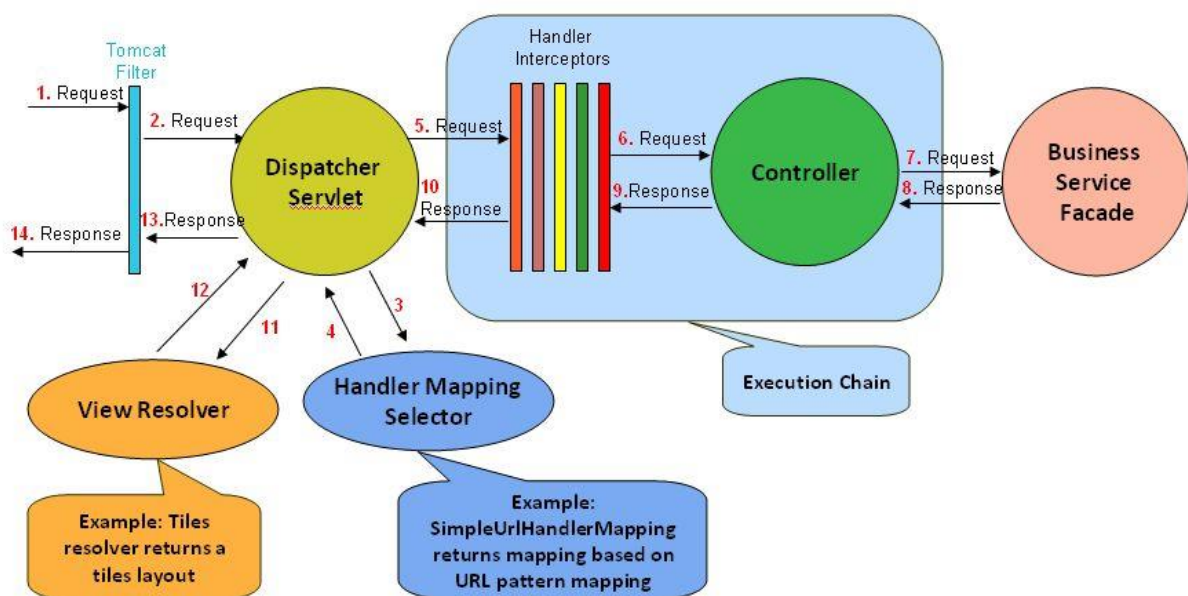
1=10
[OK]

Phương thức `$(response).each(function(index, item))` sẽ duyệt mảng response. Hàm `function(index, item)` sẽ xử lý phần tử item tại vị trí index.

5.8 Interceptor

Interceptor là một trong những thành phần quan trọng của Spring MVC. Nó cho phép chúng ta viết ra các bộ lọc để lọc các request đến các action. Sau đây là mô hình hoạt động của Interceptor.

5.8.1 Mô hình tổ chức hệ thống



Theo mô hình hoặc động ở trên thì các interceptor được đặt sau DispatcherServlet và trước các controller. Điều này có nghĩa là các yêu cầu đến action từ người dùng phải đi qua Interceptor. Tại đây chúng ta viết chương trình để xử lý yêu cầu trước khi đến action và sau khi rời action.

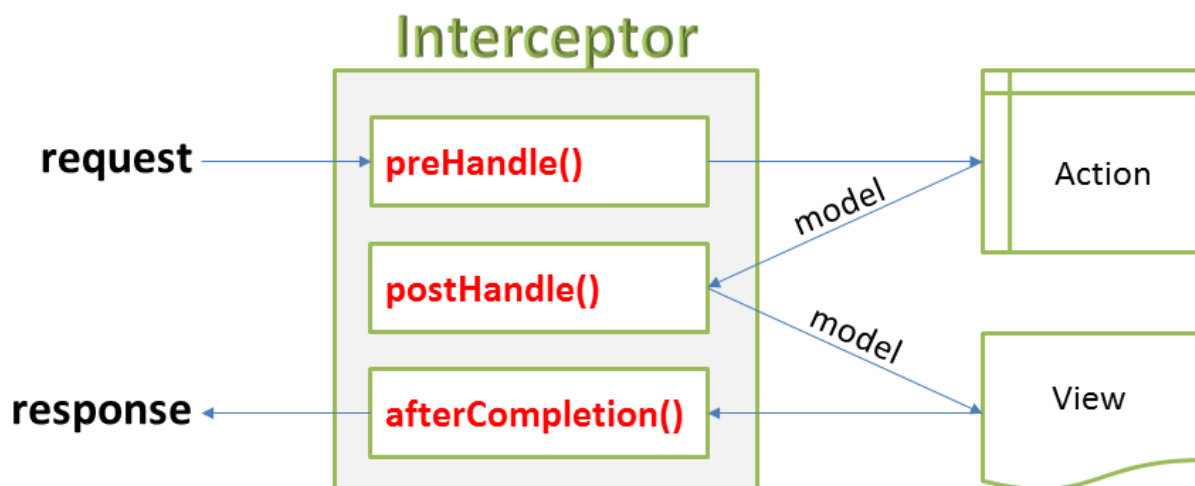
5.8.2 Cấu trúc tổ chức của Interceptor

```
public class MyInterceptor extends HandlerInterceptorAdapter{
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("preHandle()");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("postHandle()");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("afterCompletion()");
    }
}
```

Hoạt động của interceptor được mô tả như hình sau



Yêu cầu sẽ đến preHandle() trước khi đến action. Sau khi action thực hiện xong yêu cầu được trả lại cho postHandle(). Sau khi postHandle() thực hiện xong nhiệm vụ thì yêu cầu sẽ được chuyển đến view. Sau khi view kết xuất kết quả thì yêu cầu được chuyển đến afterCompletion() và sau đó trả về client.

5.8.3 Cấu hình interceptor để lọc các action

5.8.3.1 Ví dụ

Để lọc các action nào đó, bạn cần khai báo trong file cấu hình của Spring.

Giả sử bạn muốn lọc action home/index.htm của HomeController sau

```
@Controller
@RequestMapping("home")
public class HomeController {

    @RequestMapping("index")
    public String index() {
        System.out.println("HomeController.index()");
        return "index";
    }

    @RequestMapping("about")
    public String about() {
        System.out.println("HomeController.about()");
        return "about";
    }

    @RequestMapping("contact")
    public String contact() {
        System.out.println("HomeController.contact()");
        return "contact";
    }

    @RequestMapping("feedback")
    public String feedback() {
        System.out.println("HomeController.feedback()");
        return "feedback";
    }
}
```

Bạn có thể khai báo trong file cấu hình đoạn xml sau

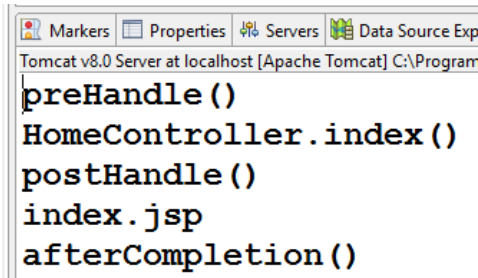
```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/home/index.htm"/>
        <bean class="nhatnghe.interceptor.MyInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

Giả sử view index.jsp có mã nguồn như sau

```
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Spring MVC - Hello</title>
</head>
<body>
    <h1>Home Page</h1>
    <%
        System.out.println("index.jsp");
    %>
```

```
</body>
</html>
```

Thì sau khi yêu cầu home/index.htm bạn sẽ quan sát thấy kết quả được xuất ở môi trường Console của eclipse như sau:



```
preHandle()
HomeController.index()
postHandle()
index.jsp
afterCompletion()
```

Kết quả này cho thấy luồng xử lý của action đã trình bày ở trên là đúng đắn.

5.8.3.2 Cấu hình Interceptor

Trên đây chỉ là một ví dụ đơn giản. Chúng ta cần xem xét việc cấu hình Interceptor một cách toàn diện hơn

- ✓ Để lọc tất cả các yêu cầu đến tất cả các action bạn chỉ cần khai báo

```
<mvc:interceptors>
    <bean class="nhatnghe.interceptor.MyInterceptor" />
</mvc:interceptors>
```

- ✓ Lọc các action có tên bắt đầu bởi /home/

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/home/**"/>
            <bean class="nhatnghe.interceptor.MyInterceptor" />
        </mvc:interceptor>
</mvc:interceptors>
```

/home/** có nghĩa là tất cả các action bắt đầu bởi home. Ví dụ home/index.htm, home/about.htm...

- ✓ Loại bỏ bớt một số action

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/home/**"/>
            <mvc:exclude-mapping path="/home/contact.htm"/>
            <mvc:exclude-mapping path="/home/about.htm"/>
            <bean class="nhatnghe.interceptor.MyInterceptor" />
        </mvc:interceptor>
</mvc:interceptors>
```

Thẻ <mvc:exclude-mapping path="/home/about.htm"/> được sử dụng để loại bỏ (không lọc) action home/about.htm.

5.8.4 Xây dựng SecurityInterceptor

Giả sử chúng ta có 2 controller là MemberController và OrderController định nghĩa các action như sau.

MemberController

```
@Controller
@RequestMapping("member")
public class MemberController {
    @RequestMapping("register")
    public String register() {
        return "register";
    }

    @RequestMapping("login")
    public String login() {
        return "login";
    }

    @RequestMapping("forgot-password")
    public String forgot() {
        return "forgot";
    }

    @RequestMapping("logoff")
    public String logoff() {
        return "logoff";
    }

    @RequestMapping("change-password")
    public String change() {
        return "change";
    }

    @RequestMapping("edit-profile")
    public String edit() {
        return "change";
    }

    @RequestMapping("activate")
    public String activate() {
        return "activate";
    }
}
```

OrderController

```
@Controller
@RequestMapping("order")
public class OrderController {
    @RequestMapping("checkout")
    public String checkout() {
        return "checkout";
    }

    @RequestMapping("my-order-list")
    public String list() {
        return "order-list";
    }
}
```

```

@RequestMapping("my-order-detail")
public String detail() {
    return "order-detail";
}

@RequestMapping("my-product-list")
public String products() {
    return "product-list";
}
}

```

Chúng ta thấy rằng chỉ có member/register.htm, member/login.htm, member/forgot-password.htm và member/activate.htm là các action được phép truy cập khi người dùng chưa đăng nhập. Các action còn lại thì phải đăng nhập mới được phép truy xuất. Nếu chưa đăng nhập mà truy xuất đến các action này thì sẽ chuyển sang action login.

Để thực hiện yêu cầu này, bạn cần xây dựng một Interceptor và cấu hình nó như sau

SecurityInterceptor sẽ làm nhiệm vụ kiểm tra xem trong session đã có attribute "user" hay chưa. Nếu chưa có (chưa đăng nhập) thì chuyển về action đăng nhập

```

public class SecurityInterceptor extends HandlerInterceptorAdapter{
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception
    {
        HttpSession session = request.getSession();
        if(session.getAttribute("user") == null){
            String path = request.getContextPath();

            // Chuyển về trang login.htm
            response.sendRedirect(path + "/member/login.htm");
            return false;
        }
        return true;
    }
}

```

Cấu hình lọc tất cả các action bắt đầu bởi /order/ và /member/ ngoại trừ 4 action đã nêu ở trên.

```

<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/member/**"/>
        <mvc:mapping path="/order/**"/>
        <mvc:exclude-mapping path="/member/register.htm"/>
        <mvc:exclude-mapping path="/member/login.htm"/>
        <mvc:exclude-mapping path="/member/forgot-password.htm"/>
        <mvc:exclude-mapping path="/member/activate.htm"/>
        <bean class="nhatnghe.interceptor.SecurityInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>

```

Với SecurityInterceptor được cấu hình như trên bạn sẽ giải quyết được mong muốn đã đặt ra.