[Bài đọc] Model 2 và mẫu thiết kế MVC

*Nguồn:*[*Spring MVC: A Tutorial (Second Edition)*](https://books.google.com.vn/books?id=O1LzCwAAQBAJ&printsec=frontcover#v=onepage&q&f=false)*, Paul Deck*

Có hai mô hình được sử dụng trong thiết kế ứng dụng web với Java, đó là Model 1 và Model 2. Model 1 là page-centric (lấy trang làm trung tâm) và chỉ phù hợp với những ứng dụng rất nhỏ. Model 2 là kiến trúc được đề xuất cho tất cả các ứng dụng, nhất là ứng dụng web với Java.

Phần này thảo luận chi tiết về Model 2 và cung cấp bốn ứng dụng mẫu với Model 2. Ví dụ đầu tiên là một ứng dụng Model 2 cơ bản với controller là một servlet. Ví dụ thứ hai cũng là một ứng dụng Model 2 đơn giản, tuy nhiên nó sử dụng một filter (bộ lọc) như là controller. Ví dụ thứ ba sử dụng một thành phần validator để xác nhận sự hợp lệ dữ liệu đầu vào của người dùng. Ứng dụng thứ tư minh họa việc sử dụng dependency injection(tiêm phụ thuộc). Trên thực tế, bạn nên sử dụng Spring.

**Tổng quan về Model 1**

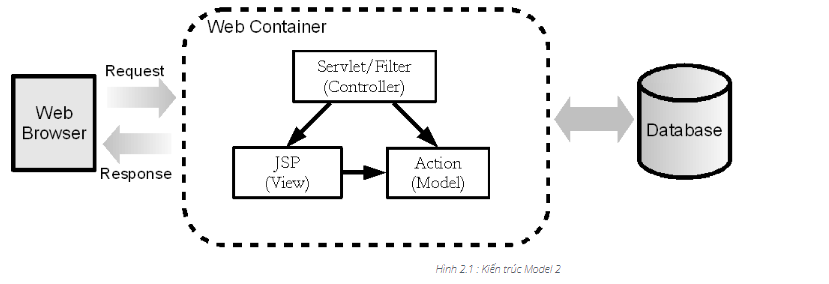
Khi bạn tìm hiểu JSP, các ứng dụng mẫu của bạn thường sẽ cho phép điều hướng từ trang này sang trang khác bằng cách cung cấp liên kết. Mặc dù phương pháp điều hướng này rất đơn giản, nhưng với các ứng dụng có quy mô trung bình hoặc lớn với số lượng trang đáng kể, việc bảo trì sẽ gặp rất nhiều khó khăn. Khi thay đổi tên của một trang JSP, có thể buộc bạn phải đổi tên các liên kết đến trang trong nhiều trang khác. Như vậy, Model 1 không được khuyến nghị sử dụng, trừ khi ứng dụng của bạn chỉ có vài ba trang.

**Tổng quan về Model 2**

Model 2 dựa trên mẫu thiết kế MVC (Model-View-Controller). Một ứng dụng áp dụng MVC bao gồm ba mô-đun: model, view và controller. View sẽ xử lý hiển thị trên ứng dụng. Model sẽ đóng gói dữ liệu của ứng dụng và lô-gic nghiệp vụ. Controller chịu trách nhiệm nhận các yêu cầu\mệnh lệnh từ người dùng tới model/view để thực hiện các thay đổi tương ứng.

Trong Model 2, bạn có một servlet hoặc một bộ lọc (filter) hoạt động với vai trò controller. Tất cả các framework web hiện đại đều triển khai theo Model 2. Các framework như Struts 1, JavaServer Faces và Spring MVC sử dụng một controller servlet trong các kiến ​​trúc MVC của mình, trong khi framework phổ biến khác là Struts 2, sử dụng một filter. Nói chung, các trang JSP được sử dụng như các khung nhìn (view) của ứng dụng, mặc dù các công nghệ hiển thị khác được hỗ trợ. Đối với model, bạn sử dụng POJO (Plain Old Java Object). POJO là các đối tượng bình thường, trái ngược với Enterprise JavaBeans (EJB) hoặc các đối tượng đặc biệt khác. Nhiều người chọn sử dụng JavaBeans (plain JavaBeans, không phải EJB) để lưu trữ các trạng thái của các đối tượng  và chuyển lô-gic nghiệp vụ tới các lớp action.

Hình 2.1 cho thấy kiến trúc của một ứng dụng Model 2:



Trong một ứng dụng Model 2, mọi yêu cầu HTTP phải được chuyển tới controller. URI (Uniform Request Identifier) nói cho controller biết action (hành động) nào cần thực hiện. Thuật ngữ “action” đề cập đến một hành động mà ứng dụng có thể thực hiện. Đối tượng Java được liên kết với một hành động được gọi là một đối tượng hành động. Một lớp action đơn lẻ có thể được sử dụng để phục vụ các hành động khác nhau (như trong Struts 2 và Spring MVC)  hoặc một hành động đơn lẻ (như trong Struts 1). Một thao tác dường như bình thường có thể thực hiện nhiều hành động. Ví dụ, việc thêm một sản phẩm vào cơ sở dữ liệu sẽ yêu cầu hai hành động:

1. Hiển thị form “Add Product” để người dùng nhập thông tin sản phẩm.
2. Lưu thông tin sản phẩm vào cơ sở dữ liệu.

Như đã đề cập ở trên, bạn sử dụng URI để báo cho controller biết action nào cần gọi. Ví dụ: để ứng dụng gửi form “Add Product”, bạn sẽ sử dụng URI như sau:

[http](https://james.codegym.vn/mod/page/view.php?id=6839)://domain/appName/input-product

 Để ứng dụng lưu một “Product”, URI sẽ là:

[http](https://james.codegym.vn/mod/page/view.php?id=6839)://domain/appName/save-product

Controller kiểm tra URI để quyết định action nào cần gọi. Nó cũng lưu trữ đối tượng model ở một nơi có thể được truy cập từ view, để các giá trị phía máy chủ có thể được hiển thị trên trình duyệt. Cuối cùng controller sử dụng một RequestDispatcher hoặc HttpServletResponse.sendRedirect () để chuyển tiếp/chuyển hướng đến một view (một trang JSP hoặc một tài nguyên khác). Trong trang JSP, bạn sử dụng các thẻ (tags) để hiển thị các giá trị.

Lưu ý, việc gọi RequestDispatcher.forward () hoặc HttpServletResponse.sendRedirect () không ngăn đoạn mã bên dưới nó được thực thi. Do đó, trừ khi lời gọi các phương thức đó là dòng mã cuối cùng, bạn cần phải trả về một cách rõ ràng.

if (action.equals(...)) {  
 RequestDispatcher rd = request.getRequestDispatcher(dispatchUrl);  
 rd.forward(request, response);  
 return;//explicitly return. Or else, the code below will be executed  
 }  
// do something else

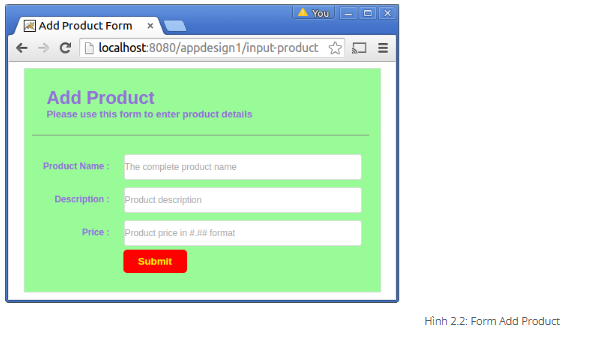
Hầu hết thời gian, bạn sẽ sử dụng một RequestDispatcher để chuyển tiếp (forward) đến một view vì nó nhanh hơn sendRedirect. Điều này là do chuyển hướng (redirect) khiến máy chủ gửi mã trạng thái phản hồi HTTP 302 với một Location Header chứa URL mới cho trình duyệt. Khi nhận được mã trạng thái 302, trình duyệt tạo một yêu cầu HTTP mới cho URL được tìm thấy trong Location Header.

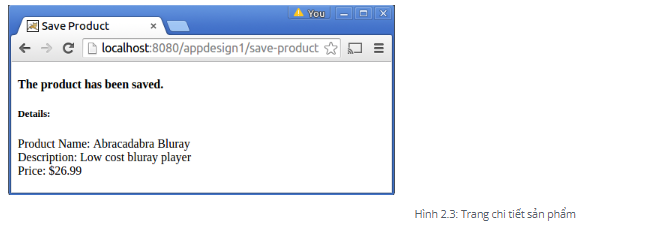
Lợi thế của việc sử dụng chuyển hướng thông qua chuyển tiếp là gì? Với chuyển hướng, bạn có thể hướng trình duyệt đến một ứng dụng khác. Bạn không thể làm điều này với một chuyển tiếp. Nếu chuyển hướng được sử dụng để truy xuất một tài nguyên khác trong cùng ứng dụng, nó tạo ra một URL khác với URL yêu cầu ban đầu. Do đó, nếu người dùng vô tình nhấn nút Tải lại/Làm mới trình duyệt sau khi phản hồi được hiển thị, mã được liên kết với URL yêu cầu ban đầu sẽ không được thực hiện lại. Ví dụ, bạn sẽ không muốn, chạy lại đoạn mã khiến thanh toán bằng thẻ tín dụng được thực hiện lại chỉ vì người dùng vô tình nhấn nút Tải lại hoặc Làm mới trên trình duyệt của mình.

**Model 2 với controller là một Servlet**

Phần này mô tả một ứng dụng Model 2 đơn giản để cung cấp ý tưởng chung về ứng dụng Model 2. Trên thực tế, các ứng dụng Model 2 sẽ phức tạp hơn nhiều.

Ứng dụng này có tên là appdesign1 được sử dụng để thêm mới sản phẩm. Người dùng điền vào một form như trong Hình 2.2 và gửi đi. Sau đó, ứng dụng sẽ trả về một trang xác nhận tới người dùng và hiển thị chi tiết của sản phẩm đã lưu. (Xem Hình 2.3)





Ứng dụng có khả năng thực hiện hai hành động sau:

1. Hiển thị form “Add Product”. Hành động này gửi form nhập trong như trong Hình 2.2 đến trình duyệt. Để URI gọi hành động này, nó phải chứa chuỗi đầu vào của sản phẩm (input).

2. Lưu sản phẩm và trả lại trang xác nhận như trong Hình 2.3. Để URI gọi hành động này, nó phải chứa chuỗi đầu ra sản phẩm (output).

Ứng dụng bao gồm các thành phần sau:

1. Lớp Product làm model. Đối tượng của lớp Product chứa thông tin sản phẩm.

2. Lớp ProductForm, bao gồm các trường của form HTML để thêm mới sản phẩm. Các thuộc tính của ProductForm được sử dụng để cung cấp thông tin cho đối tượng Product.

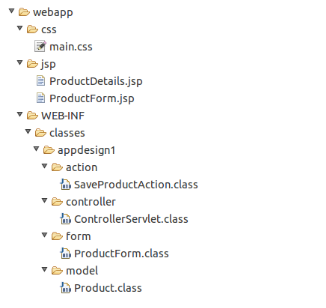
3. Lớp ControllerServlet, đảm nhiệm vai trò controller của ứng dụng.

4. Lớp action có tên là SaveProductAction.

5. Hai trang JSP (ProductForm.jsp và ProductDetails.jsp) là các view.

6. Một tệp CSS định kiểu cho các các view.

Cấu trúc thư mục của ứng dụng này được hiển thị trong Hình 2.4.



Hình 2.4: Cấu trúc thư mục của ứng dụng

Chúng ta hãy xem xét kỹ hơn từng thành phần trong ứng dụng appdesign1.

Lớp Product

Một thể hiện (đối tượng) của lớp Product là một JavaBean chứa các thông tin về sản phẩm. Lớp Product (được hiển thị trong VD 2.1) có ba thuộc tính: productName, description và price.

VD 2.1: Lớp Product

package appdesign1.model;  
import java.io.Serializable;  
import java.math.BigDecimal;  
  
public class Product implements Serializable {  
 private static final long *serialVersionUID* = 748392348L;  
 private String name;  
 private String description;  
 private BigDecimal price;  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getDescription() {  
 return description;  
 }  
 public void setDescription(String description) {  
 this.description = description;  
 }  
 public BigDecimal getPrice() {  
 return price;  
 }  
 public void setPrice(BigDecimal price) {  
 this.price = price;  
 }  
}

Lớp Product triển khai java.io.Serializable để các thể hiện của nó có thể được lưu trữ an toàn trong các đối tượng HttpSession. Khi triển khai Serializable, Product phải có thuộc tính serialVersionUID.

**Lớp ProductForm**

Lớp này ánh xạ tới một form HTML. Nó đại diện cho form HTML trên máy chủ. Lớp ProductForm, trong VD 2.2, chứa các thông tin của một sản phẩm dưới dạng chuỗi. Thoạt nhìn, lớp ProductForm tương tự như lớp Product và bạn có thể hỏi tại sao cần lớp ProductForm. Một đối tượng form lưu các ServletRequest vào những thành phần khác, chẳng hạn như validators. ServletRequest là một kiểu servlet cụ thể và không được tiếp xúc với các tầng khác của ứng dụng.

Mục đích thứ hai của đối tượng form là giữ lại trạng thái của form nếu dữ liệu đầu vào không hợp lệ.

Lưu ý, lớp form không phải triển khai interface Serializable, các đối tượng này cũng hiếm khi được lưu trữ trong một HttpSession.

VD 2.2: Lớp ProductForm

package appdesign1.form;  
public class ProductForm {  
 private String name;  
 private String description;  
 private String price;  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getDescription() {  
 return description;  
 }  
 public void setDescription(String description) {  
 this.description = description;  
 }  
 public String getPrice() {  
 return price;  
 }  
 public void setPrice(String price) {  
 this.price = price;  
 }  
}

**Lớp ControllerServlet**

Lớp ControllerServlet (được trình bày trong VD 2.3) kế thừa lớp javax.servlet.http.HttpServlet. Cả hai phương thức doGet và doPost của lớp này gọi phương thức xử lý, là bộ não của controller servlet.

VD 2.3: Lớp ControllerServlet

package appdesign1.controller;  
import java.io.IOException;  
import javax.servlet.RequestDispatcher;  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import appdesign1.action.SaveProductAction;  
import appdesign1.form.ProductForm;  
import appdesign1.model.Product;  
import java.math.BigDecimal;  
  
@WebServlet(name = "ControllerServlet", urlPatterns = {  
 "/input-product", "/save-product" })  
public class ControllerServlet extends HttpServlet {  
  
 private static final long *serialVersionUID* = 1579L;  
  
 @Override  
 public void doGet(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
 process(request, response);  
 }  
  
 @Override  
 public void doPost(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
 process(request, response);  
 }  
  
 private void process(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
  
 String uri = request.getRequestURI();  
 /\*  
 \* uri is in this form: /contextName/resourceName,  
 \* for example: /appdesign1/input-product.  
 \* However, in the event of a default context, the  
 \* context name is empty, and uri has this form  
 \* /resourceName, e.g.: /input-product  
 \*/  
 int lastIndex = uri.lastIndexOf("/");  
 String action = uri.substring(lastIndex + 1);  
 // execute an action  
 String dispatchUrl = null;  
 if ("input-product".equals(action)) {  
 // no action class, just forward  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 } else if ("save-product".equals(action)) {  
 // create form  
 ProductForm productForm = new ProductForm();  
 // populate action properties  
 productForm.setName(request.getParameter("name"));  
 productForm.setDescription(  
 request.getParameter("description"));  
 productForm.setPrice(request.getParameter("price"));  
  
 // create model  
 Product product = new Product();  
 product.setName(productForm.getName());  
 product.setDescription(productForm.getDescription());  
 try {  
 product.setPrice(new BigDecimal(productForm.getPrice()));  
 } catch (NumberFormatException e) {  
 }  
 // execute action method  
 SaveProductAction saveProductAction =  
 new SaveProductAction();  
 saveProductAction.save(product);  
  
 // store model in a scope variable for the view  
 request.setAttribute("product", product);  
 dispatchUrl = "/jsp/ProductDetails.jsp";  
 }  
  
 if (dispatchUrl != null) {  
 RequestDispatcher rd =  
 request.getRequestDispatcher(dispatchUrl);  
 rd.forward(request, response);  
 }  
 }  
}

Phương thức xử lý trong lớp ControllerServlet xử lý tất cả các yêu cầu gửi đến. Nó bắt đầu bằng cách lấy URI và tên action.

String uri = request.getRequestURI();  
 int lastIndex = uri.lastIndexOf("/");  
 String action = uri.substring(lastIndex + 1);

Giá trị của action trong ứng dụng này có thể là sản phẩm đầu vào (**input-product**) hoặc sản phẩm đã lưu (**save-product**).

Phương thức xử lý sau đó tiếp tục bằng cách thực hiện các bước sau:

1. Khởi tạo một lớp action liên quan, nếu có.

2. Nếu một đối tượng action tồn tại, hãy tạo và cung cấp thông tin cho một đối tượng form với các tham số của request. Có ba thuộc tính trong action **save-product**: name, description và price. Tiếp theo, tạo một đối tượng model và cung cấp giá trị cho các thuộc tính của nó từ đối tượng form.

3. Nếu một đối tượng action tồn tại, hãy gọi phương thức action.

4. Chuyển tiếp yêu cầu đến một view (trang JSP).

Phương thức process xác định action cần thực hiện là gì, nếu như sau:

// execute an action  
 if ("input-product".equals(action)) {  
 // no action class, just forward  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 } else if ("save-product".equals(action)) {  
 // instantiate action class  
 ...  
 }

Không có lớp action nào để khởi tạo cho đầu vào của action. Đối với product đã lưu, phương thức process tạo ra một Product, một ProductForm và các giá trị được sao chép từ form. Ở giai đoạn này, không đảm bảo tất cả các thuộc tính là khác kiểu chuỗi, chẳng hạn như giá, nhưng chúng ta sẽ giải quyết vấn đề này sau trong phần kiểm tra tính hợp lệ của form (Validators). Phương thức process sau đó khởi tạo lớp SaveProductAction và gọi phương thức save của nó.

// create form  
 ProductForm productForm = new ProductForm();  
 // populate action properties  
 productForm.setName(request.getParameter("name"));  
 productForm.setDescription(  
 request.getParameter("description"));  
 productForm.setPrice(request.getParameter("price"));  
  
 // create model  
 Product product = new Product();  
 product.setName(productForm.getName());  
 product.setDescription(productForm.getDescription());  
 try {  
 product.setPrice(new BigDecimal(productForm.getPrice()));  
 } catch (NumberFormatException e) {  
 }  
 // execute action method  
 SaveProductAction saveProductAction =  
 new SaveProductAction();  
 saveProductAction.save(product);

Product sau đó được lưu trữ trong HttpServletRequest sao cho view có thể truy cập nó.

// store action in a scope variable for the view  
 request.setAttribute("product", product);

Phương thức process kết thúc bằng cách chuyển tiếp đến một view. Nếu action là input-product, điều khiển được chuyển tiếp đến trang ProductForm.jsp. Nếu action là save-product, điều khiển được chuyển tiếp đến trang ProductDetails.jsp.

// forward to a view  
 if (dispatchUrl != null) {  
 RequestDispatcher rd =  
 request.getRequestDispatcher(dispatchUrl);  
 rd.forward(request, response);  
 }

**Lớp Action**

Chỉ có một lớp action trong ứng dụng, có trách nhiệm lưu một product vào nơi lưu trữ, chẳng hạn như cơ sở dữ liệu. Lớp action được đặt tên là SaveProductAction và được đưa ra trong VD 2.4.

VD 2.4: Lớp SaveProductAction

package appdesign1.action;  
  
public class SaveProductAction {  
 public void save(Product product) {  
 // insert Product to the database  
 }  
}

Trong ví dụ này, lớp SaveProductAction không thi hành phương thức save.

**Views**

Ứng dụng sử dụng hai trang JSP cho các view. Trang đầu tiên, ProductForm.jsp, được hiển thị nếu action là product input. Trang thứ hai, ProductDetails.jsp, được hiển thị cho product output. ProductForm.jsp được đưa ra trong VD 2.5 và ProductDetails.jsp trong VD 2.6.

VD 2.5: Trang ProductForm.jsp

<!DOCTYPE html>  
<html>  
<head>  
<title>Add Product Form</title>  
<style type="text/css">@import url(css/main.css);</style>  
</head>  
<body>  
<form method="post" action="save-product">  
<h1>Add Product  
<span>Please use this form to enter product details</span>  
</h1>  
<label>  
<span>Product Name :</span>  
<input id="name" type="text" name="name"  
 placeholder="The complete product name"/>  
</label>  
<label>  
<span>Description :</span>  
<input id="description" type="text" name="description"  
 placeholder="Product description"/>  
</label>  
<label>  
<span>Price :</span>  
<input id="price" name="price" type="number" step="any"  
 placeholder="Product price in #.## format"/>  
</label>  
<label>  
<span>&nbsp;</span>  
<input type="submit"/>  
</label>  
</form>  
</body>  
</html>

**chú ý**

Không sử dụng HTML table để định dạng form. Thay vào đó, hãy sử dụng CSS.

**chú thích**

Thuộc tính step trong trường nhập giá buộc trình duyệt cho phép nhập số thập phân.

VD 2.6: Trang ProductDetails.jsp

<!DOCTYPE html>  
<html>  
<head>  
<title>Save Product</title>  
<style type="text/css">@import url(css/main.css);</style>  
</head>  
<body>  
<div id="global">  
<h4>The product has been saved.</h4>  
<p>  
<h5>Details:</h5>  
 Product Name: ${product.name}<br/>  
 Description: ${product.description}<br/>  
 Price: $${product.price}  
</p>  
</div>  
</body>  
</html>

Trang ProductForm.jsp chứa một form HTML để nhập chi tiết sản phẩm. Trang ProductDetails.jsp sử dụng Ngôn ngữ biểu thức (EL) để truy cập vào đối tượng product trong HttpServletRequest.

Trong ứng dụng này, như trường hợp của hầu hết các ứng dụng Model 2, bạn cần phải ngăn các trang JSP được truy cập trực tiếp từ trình duyệt. Có một số cách để đạt được điều này, bao gồm:

* Đưa các trang JSP vào WEB-INF. Mọi thứ trong WEB-INF hoặc thư mục con trong WEB-INF đều được bảo vệ. Nếu bạn đặt các trang JSP của mình trong WEB-INF, bạn không thể truy cập chúng trực tiếp từ trình duyệt, nhưng controller vẫn có thể gửi yêu cầu đến các trang đó. Tuy nhiên, đây không phải là cách tiếp cận được khuyến khích vì không phải tất cả các vùng chứa đều triển khai tính năng này.
* Sử dụng filter servlet và lọc ra các yêu cầu cho các trang JSP.
* Sử dụng hạn chế bảo mật trong bộ mô tả triển khai. Điều này dễ hơn việc sử dụng filter vì bạn không phải viết class filter. Phương pháp này được chọn cho ứng dụng này.

**Testing**

Giả sử bạn đang chạy ứng dụng trên máy cục bộ của mình trên cổng 8080, bạn có thể gọi ứng dụng bằng cách sử dụng URL sau:

<http://localhost:8080/appdesign1/input-product>

Khi bạn gửi form, URL sau sẽ được gửi đến máy chủ:

<http://localhost:8080/appdesign1/save-product>

Sử dụng controller servlet cho phép bạn sử dụng servlet như một trang welcome. Đây là một tính năng quan trọng vì sau đó bạn có thể cấu hình ứng dụng của mình để controller servlet sẽ được gọi bằng cách nhập tên miền của bạn (chẳng hạn như http://example.com) vào thanh địa chỉ của trình duyệt. Bạn không thể thực hiện việc này bằng filter.

**Model 2 với một Filter Dispatcher**

Trong khi một servlet là controller phổ biến nhất trong một ứng dụng Model 2, một filter cũng có thể hoạt động như một controller. Tuy nhiên, lưu ý rằng filter không có đặc quyền hoạt động như một trang welcome. Chỉ cần nhập tên miền sẽ không gọi điều phối filter. Struts 2 sử dụng filter làm controller vì filter cũng được sử dụng để phân phối nội dung tĩnh.

Ví dụ sau (appdesign2) là một ứng dụng Model 2 sử dụng bộ điều phối filter. Cấu trúc thư mục của appdesign2 được thể hiện trong Hình 2.5.

Hình 2.5: Cấu trúc thư mục của appdesign2

Các trang JSP và lớp Product giống với các trang trong appdesign1. Tuy nhiên, thay vì một servlet làm controller, bạn có một filter gọi là FilterDispatcher ( trong VD 2.7).

VD 2.7: Lớp DispatcherFilter

package appdesign2.filter;  
import java.io.IOException;  
import javax.servlet.Filter;  
import javax.servlet.FilterChain;  
import javax.servlet.FilterConfig;  
import javax.servlet.RequestDispatcher;  
import javax.servlet.ServletException;  
import javax.servlet.ServletRequest;  
import javax.servlet.ServletResponse;  
import javax.servlet.annotation.WebFilter;  
import javax.servlet.http.HttpServletRequest;  
import appdesign2.action.SaveProductAction;  
import appdesign2.form.ProductForm;  
import appdesign2.model.Product;  
import java.math.BigDecimal;  
  
@WebFilter(filterName = "DispatcherFilter",  
 urlPatterns = { "/\*" })  
public class DispatcherFilter implements Filter {  
  
 @Override  
 public void init(FilterConfig filterConfig)  
 throws ServletException {  
 }  
  
 @Override  
 public void destroy() {  
 }  
  
 @Override  
 public void doFilter(ServletRequest request,  
 ServletResponse response, FilterChain filterChain)  
 throws IOException, ServletException {  
 HttpServletRequest req = (HttpServletRequest) request;  
 String uri = req.getRequestURI();  
 /\*  
 \* uri is in this form: /contextName/resourceName, for  
 \* example /appdesign2/input-product. However, in the  
 \* case of a default context, the context name is empty,  
 \* and uri has this form /resourceName, e.g.:  
 \* /input-product  
 \*/  
 // action processing  
 int lastIndex = uri.lastIndexOf("/");  
 String action = uri.substring(lastIndex + 1);  
 String dispatchUrl = null;  
 if ("input-product".equals(action)) {  
 // do nothing  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 } else if ("save-product".equals(action)) {  
 // create form  
 ProductForm productForm = new ProductForm();  
 // populate action properties  
 productForm.setName(request.getParameter("name"));  
 productForm.setDescription(  
 request.getParameter("description"));  
 productForm.setPrice(request.getParameter("price"));  
  
 // create model  
 Product product = new Product();  
 product.setName(productForm.getName());  
 product.setDescription(product.getDescription());  
 try {  
 product.setPrice(new BigDecimal(productForm.getPrice()));  
 } catch (NumberFormatException e) {  
 }  
 // execute action method  
 SaveProductAction saveProductAction =  
 new SaveProductAction();  
 saveProductAction.save(product);  
  
 // store model in a scope variable for the view  
 request.setAttribute("product", product);  
 dispatchUrl = "/jsp/ProductDetails.jsp";  
 }  
  
 // forward to a view  
 if (dispatchUrl != null) {  
 RequestDispatcher rd = request  
 .getRequestDispatcher(dispatchUrl);  
 rd.forward(request, response);  
 } else {  
 // let static contents pass  
 filterChain.doFilter(request, response);  
 }  
 }  
}

Phương thức doFilter thực hiện những gì phương thức process trong appdesign1 đã làm.

Vì filter nhắm mục tiêu tất cả các URL include nội dung tĩnh, bạn cần gọi filterChain.doFilter () nếu không có action nào được gọi.

} else {  
 // let static contents pass  
 filterChain.doFilter(request, response);  
 }

Để kiểm tra ứng dụng, hãy điều hướng trình duyệt của bạn đến URL này:

<http://localhost:8080/appdesign2/input-product>

Validators

Xác thực đầu vào là một bước quan trọng trong việc thi hành một action. Phạm vi xác thực từ các tác vụ đơn giản như kiểm tra xem trường nhập liệu có giá trị hay không, đến các tác vụ phức tạp hơn như xác minh số thẻ tín dụng. Trong thực tế, do việc xác thực đóng một vai trò quan trọng như vậy nên cộng đồng Java đã xuất bản cuốn JSR 303, “Bean Validation”, để chuẩn hóa xác thực đầu vào trong Java. Các framework MVC hiện đại thường cung cấp cả hai phương thức xác nhận (lập trình và khai báo). Trong xác nhận bằng lập trình, bạn viết mã để xác thực đầu vào của người dùng. Trong xác nhận khai báo, bạn cung cấp các quy tắc xác thực trong các tài liệu XML hoặc thông qua các thuộc tính.

**chú thích**

Mặc dù bạn có thể thực hiện xác thực đầu vào phía máy khách bằng HTML5 hoặc JavaScript, chúng ta không nên  dựa vào nó vì người dùng hiểu biết có thể vượt qua nó một cách dễ dàng. Luôn thực hiện xác thực đầu vào phía máy chủ.

Ví dụ sau đây (appdesign3) mở rộng ứng dụng Mô hình 2 dựa trên controller servlet trong appdesign1. Ứng dụng kết hợp một trình xác nhận sản phẩm được đưa ra trong VD 2.8.

VD 2.8: Lớp ProductValidator

package appdesign3.validator;  
import java.util.ArrayList;  
import java.util.List;  
import appdesign3.form.ProductForm;  
  
public class ProductValidator {  
 public List<String> validate(ProductForm productForm) {  
 List<String> errors = new ArrayList<>();  
 String name = productForm.getName();  
 if (name == null || name.trim().isEmpty()) {  
 errors.add("Product must have a name");  
 }  
 String price = productForm.getPrice();  
 if (price == null || price.trim().isEmpty()) {  
 errors.add("Product must have a price");  
 } else {  
 try {  
 Float.*parseFloat*(price);  
 } catch (NumberFormatException e) {  
 errors.add("Invalid price value");  
 }  
 }  
 return errors;  
 }  
}

Lớp ProductValidator trong VD 2.8 cung cấp phương thức xác nhận hợp lệ hoạt động trên ProductForm. Trình xác thực đảm bảo rằng, sản phẩm có name không trống và price của nó là một số hợp lệ. Phương thức validate trả về một danh sách các chuỗi chứa các thông báo lỗi xác thực. Danh sách trống có nghĩa là xác thực thành công.

Bây giờ bạn có một trình xác nhận hợp lệ, bạn cần báo cho controller sử dụng nó. VD 2.9 trình bày phiên bản sửa đổi của ControllerServlet. Chú ý đến các dòng in đậm.

VD 2.9: Lớp ControllerServlet trong appdesign3

package appdesign3.controller;  
  
import java.io.IOException;  
import java.util.List;  
import javax.servlet.RequestDispatcher;  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import appdesign3.action.SaveProductAction;  
import appdesign3.form.ProductForm;  
import appdesign3.model.Product;  
import appdesign3.validator.ProductValidator;  
import java.math.BigDecimal;  
  
@WebServlet(name = "ControllerServlet", urlPatterns = {  
 "/input-product", "/save-product" })  
public class ControllerServlet extends HttpServlet {  
  
 private static final long *serialVersionUID* = 98279L;  
  
 @Override  
 public void doGet(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
 process(request, response);  
 }  
  
 @Override  
 public void doPost(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
 process(request, response);  
 }  
  
 private void process(HttpServletRequest request,  
 HttpServletResponse response)  
 throws IOException, ServletException {  
  
 String uri = request.getRequestURI();  
 /\*  
 \* uri is in this form: /contextName/resourceName,  
 \* for example: /appdesign1/input-product.  
 \* However, in the case of a default context, the  
 \* context name is empty, and uri has this form  
 \* /resourceName, e.g.: /input-product  
 \*/  
 int lastIndex = uri.lastIndexOf("/");  
 String action = uri.substring(lastIndex + 1);  
 String dispatchUrl = null;  
  
 if ("input-product".equals(action)) {  
 // no action class, there is nothing to be done  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 } else if ("save-product".equals(action)) {  
 // instantiate action class  
 ProductForm productForm = new ProductForm();  
 // populate action properties  
 productForm.setName(  
 request.getParameter("name"));  
 productForm.setDescription(  
 request.getParameter("description"));  
 productForm.setPrice(request.getParameter("price"));  
  
 // validate ProductForm  
 ProductValidator productValidator = new  
 ProductValidator();  
 List<String> errors =  
 productValidator.validate(productForm);  
 if (errors.isEmpty()) {  
 // create Product from ProductForm  
 Product product = new Product();  
 product.setName(productForm.getName());  
 product.setDescription(  
 productForm.getDescription());  
 product.setPrice(new BigDecimal(productForm.getPrice()));  
  
 // no validation error, execute action method  
 SaveProductAction saveProductAction = new  
 SaveProductAction();  
 saveProductAction.save(product);  
  
 // store action in a scope variable for the view  
 request.setAttribute("product", product);  
 dispatchUrl = "/jsp/ProductDetails.jsp";  
 } else {  
 request.setAttribute("errors", errors);  
 request.setAttribute("form", productForm);  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 }  
 }  
  
 // forward to a view  
 if (dispatchUrl != null) {  
 RequestDispatcher rd =  
 request.getRequestDispatcher(dispatchUrl);  
 rd.forward(request, response);  
 }  
 }  
}

Lớp ControllerServlet trong VD 2.9 chèn mã khởi tạo lớp ProductValidator và gọi phương thức xác thực của nó trên product output.

// validate ProductForm  
ProductValidator productValidator = new ProductValidator();  
 List<String> errors =  
 productValidator.validate(productForm);

Phương thức xác nhận có ProductForm, bao gồm thông tin product được nhập vào form HTML. Nếu không có ProductForm, bạn sẽ phải chuyển ServletRequest tới trình xác nhận hợp lệ.

Phương thức xác thực trả về một Danh sách trống nếu xác thực thành công, trong trường hợp đó Product được tạo và được chuyển đến một SaveProductAction. Khi xác nhận thành công, controller lưu trữ product trong ServletContext và chuyển tiếp đến trang ProductDetails.jsp, sau đó hiển thị chi tiết của product. Nếu xác nhận không thành công, controller lưu trữ danh sách lỗi ProductForm trong ServletContext và chuyển tiếp trở lại ProductForm.jsp.

if (errors.isEmpty()) {  
 // create Product from ProductForm  
 Product product = new Product();  
 product.setName(productForm.getName());  
 product.setDescription(  
 productForm.getDescription());  
 product.setPrice(new BigDecimal(productForm.getPrice()));  
  
 // no validation error, execute action method  
 SaveProductAction saveProductAction = new  
 SaveProductAction();  
 saveProductAction.save(product);  
  
 // store action in a scope variable for the view  
 request.setAttribute("product", product);  
 dispatchUrl = "/jsp/ProductDetails.jsp";  
 } else {  
 request.setAttribute("errors", errors);  
 request.setAttribute("form", productForm);  
 dispatchUrl = "/jsp/ProductForm.jsp";  
 }

Trang ProductForm.jsp trong appdesign3 đã được sửa đổi để có khả năng hiển thị các thông báo lỗi và hiển thị lại các giá trị không hợp lệ. VD 2.10 cho thấy ProductForm.jsp trong appdesign3.

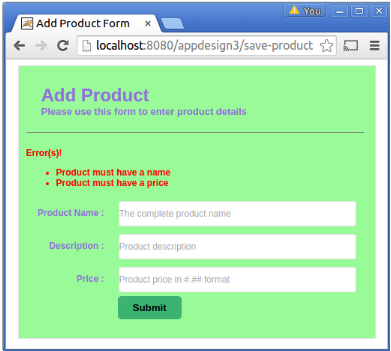
VD 2.10: Trang ProductForm.jsp trong appdesign3

<!DOCTYPE html>  
<html>  
<head>  
<title>Add Product Form</title>  
<style type="text/css">@import url(css/main.css);</style>  
</head>  
<body>  
<form method="post" action="save-product">  
<h1>Add Product  
<span>Please use this form to enter product details</span>  
</h1>  
 ${empty requestScope.errors? "" : "<p style='color:red'>"  
 += "Error(s)!"  
 += "<ul>"}  
<!--${requestScope.errors.stream().map(  
 x -> "--><li>"+=x+="</li><!--").toList()}-->  
 ${empty requestScope.errors? "" : "</ul></p>"}  
<label>  
<span>Product Name :</span>  
<input id="name" type="text" name="name"  
 placeholder="The complete product name"  
 value="${form.name}"/>  
</label>  
<label>  
<span>Description :</span>  
<input id="description" type="text" name="description"  
 placeholder="Product description"  
 value="${form.description}"/>  
</label>  
<label>  
<span>Price :</span>  
<input id="price" name="price" type="number" step="any"  
 placeholder="Product price in #.## format"  
 value="${form.price}"/>  
</label>  
<label>  
<span>&nbsp;</span>  
<input type="submit"/>  
</label>  
</form>  
</body>  
</html>

Bạn có thể kiểm tra appdesign3 bằng cách gọi action của product input:

<http://localhost:8080/appdesign3/input-product>

Nếu form Sản phẩm chứa giá trị không hợp lệ khi bạn gửi, một thông báo lỗi sẽ được hiển thị cùng với giá trị không chính xác. Hình 2.6 cho thấy hai thông báo lỗi xác thực.



Hình 2.6: thông báo lỗi xác thực

Tổng kết:

Trong phần này, bạn đã học về kiến ​​trúc Model 2, dựa trên mẫu thiết kế MVC và cách viết các ứng dụng Model 2 bằng cách sử dụng controller servlet hoặc bộ lọc filter. Hai kiểu ứng dụng Model 2 này đã được trình bày trong appdesign1 và appdesign2. Một lợi thế rõ ràng của việc sử dụng một servlet như controller trên một filter là bạn có thể cấu hình servlet như một trang welcome. Trong một ứng dụng Model 2, các trang JSP thường được sử dụng làm view, mặc dù các công nghệ khác như Apache Velocity và FreeMarker cũng có thể được sử dụng. Nếu các trang JSP được sử dụng làm view trong kiến ​​trúc Model 2, các trang đó được sử dụng để chỉ hiển thị các giá trị và không có phần tử scripting nào có trong chúng.

Trong phần này, bạn cũng đã xây dựng một framework MVC đơn giản kết hợp một trình xác nhận hợp lệ và phân phối nó với một bộ tiêm phụ thuộc. Trong khi framework tự làm là cách luyện tập tốt, thì bạn nên triển khai các dự án MVC của bạn trên một framework MVC hoàn thiện như Spring MVC.

Last modified: Friday, 22 November 2019, 3:29 PM

[Previous Activity](https://james.codegym.vn/mod/page/view.php?id=8870&forceview=1)

Top of Form

Jump to...

Bottom of Form