Chapter 4. Developing the Web Interface

This is Part 4 of a step-by-step account of how to develop a web application from scratch using the Spring Framework. In Part 1 we configured the environment and set up a basic application. In Part 2 we refined the application that we will build upon. Part 3 added all the business logic and unit tests. It's now time to build the actual web interface for the application.

4.1. Add reference to business logic in the controller

First of all, let's rename our `HelloController` to something more meaningful. How about `InventoryController` since we are building an inventory system. This is where an IDE with refactoring support is invaluable. We rename `HelloController` to `InventoryController` and the `HelloControllerTests` to `InventoryControllerTests`. Next, We modify the `InventoryController` to hold a reference to the `ProductManager` class. We also add code to have the controller pass some product information to the view. The `getModelAndView()` method now returns a `Map` with both the date and time and the products list obtained from the manager reference.

'springapp/src/springapp/web/InventoryController.java':

```
package springapp.web;

import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.util.Map;
import java.util.HashMap;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import springapp.service.ProductManager;

public class InventoryController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    private ProductManager productManager;

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String now = (new java.util.Date()).toString();
        logger.info("returning hello view with " + now);

        Map<String, Object> myModel = new HashMap<String, Object>();
        myModel.put("now", now);
        myModel.put("products", this.productManager.getProducts());

        return new ModelAndView("hello", "model", myModel);
    }


    public void setProductManager(ProductManager productManager) {
        this.productManager = productManager;
    }

}
```

We will also need to modify the InventoryControllerTests to supply a ProductManager and extract the value for `'now'` from the model Map before the tests will pass again.

'springapp/test/springapp/web/InventoryControllerTests.java':

```
package springapp.web;

import java.util.Map;

import org.springframework.web.servlet.ModelAndView;

import springapp.service.SimpleProductManager;
import springapp.web.InventoryController;

import junit.framework.TestCase;

public class InventoryControllerTests extends TestCase {

    public void testHandleRequestView() throws Exception{
        InventoryController controller = new InventoryController();
        controller.setProductManager(new SimpleProductManager());
        ModelAndView modelAndView = controller.handleRequest(null, null);
        assertEquals("hello", modelAndView.getViewName());
        assertNotNull(modelAndView.getModel());
        Map modelMap = (Map) modelAndView.getModel().get("model");
        String nowValue = (String) modelMap.get("now");
        assertNotNull(nowValue);
    }
}
```

4.2. Modify the view to display business data and add support for message bundle

Using the JSTL `<c:forEach/>` tag, we add a section that displays product information. We have also replaced the title, heading and greeting text with a JSTL `<fmt:message/>` tag that pulls the text to display from a provided `'message'` source – we will show this source in a later step.

'springapp/war/WEB-INF/jsp/hello.jsp':

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
  <head><title><fmt:message key="title"/></title></head>
  <body>
    <h1><fmt:message key="heading"/></h1>
    <p><fmt:message key="greeting"/> <c:out value="${model.now}"/></p>
    <h3>Products</h3>
    <c:forEach items="${model.products}" var="prod">
      <c:out value="${prod.description}"/> <i>$<c:out value="${prod.price}"/></i><br><br>
    </c:forEach>
  </body>
</html>
```

4.3. Add some test data to automatically populate some business objects

It's time to add a `SimpleProductManager` to our configuration file and to pass that into the setter of the `InventoryController`. We are not going to add any code to load the business objects from a database just yet. Instead, we can stub a couple of `Product` instances using Spring's bean and application context support. We will simply put the data we need as a couple of bean entries in **'springapp-servlet.xml'**. We will also add the `'messageSource'` bean entry that will pull in the messages resource bundle (**'messages.properties'**) that we will create in the next step. Also remember to rename the reference to `HelloController` to `InventoryController` since we renamed it.

'springapp/war/WEB-INF/springapp-servlet.xml':

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!-- the application context definition for the springapp DispatcherServlet -->

    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="products">
            <list>
                <ref bean="product1"/>
                <ref bean="product2"/>
                <ref bean="product3"/>
            </list>
        </property>
    </bean>

    <bean id="product1" class="springapp.domain.Product">
        <property name="description" value="Lamp"/>
        <property name="price" value="5.75"/>
    </bean>

    <bean id="product2" class="springapp.domain.Product">
        <property name="description" value="Table"/>
        <property name="price" value="75.25"/>
    </bean>

    <bean id="product3" class="springapp.domain.Product">
        <property name="description" value="Chair"/>
        <property name="price" value="22.79"/>
    </bean>

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
    </bean>

    <bean name="/hello.htm" class="springapp.web.InventoryController">
        <property name="productManager" ref="productManager"/>
    </bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

4.4. Add the message bundle and a 'clean' target to 'build.xml'

We create a **'messages.properties'** file in the **'war/WEB-INF/classes'** directory. This properties bundle so far has three entries matching the keys specified in the `<fmt:message/>` tags that we added to **'hello.jsp'**.

'springapp/war/WEB-INF/classes/messages.properties':

```
title=SpringApp
heading=Hello :: SpringApp
greeting=Greetings, it is now
```

Since we moved some source files around, it makes sense to add a `'clean'` and an `'undeploy'` target to the build script. We add the following entries to the **'build.xml'** file.

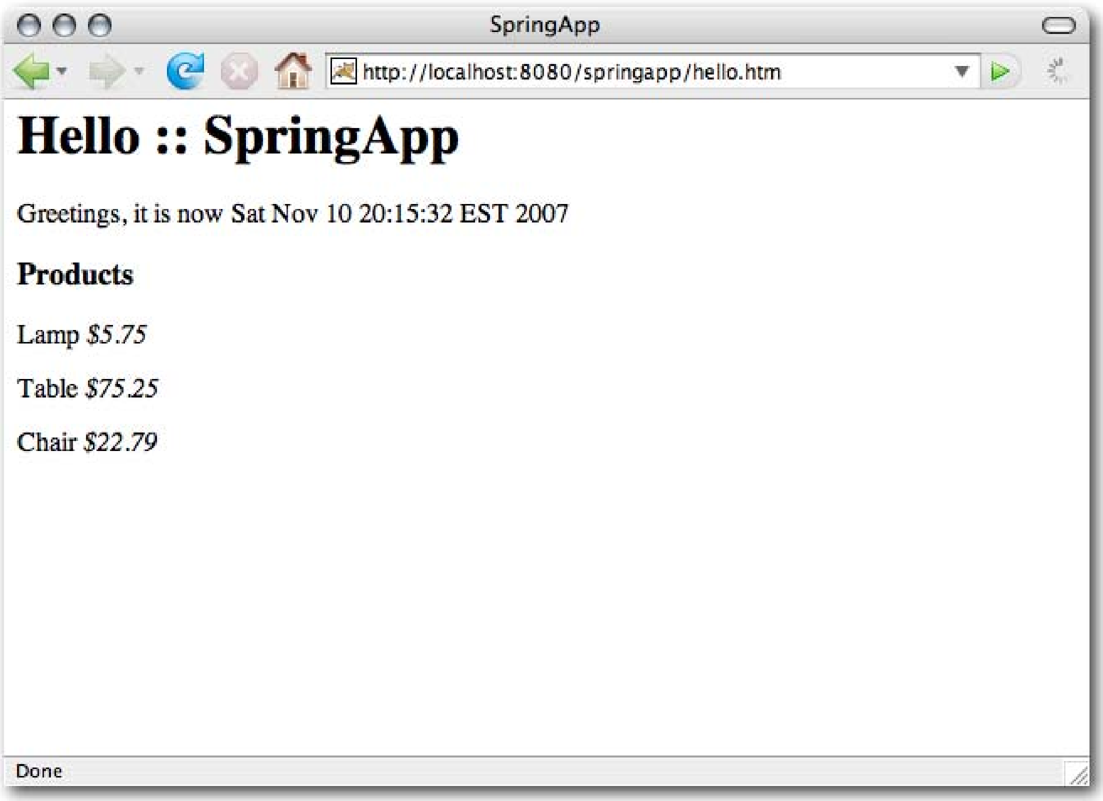
'build.xml':

```
    <target name="clean" description="Clean output directories">
        <delete>
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
            </fileset>
        </delete>
    </target>

    <target name="undeploy" description="Un-Deploy application">
        <delete>
```

```
            <fileset dir="${deploy.path}/${name}">
                <include name="**/*.*"/>
            </fileset>
        </delete>
    </target>
```

Now stop the Tomcat server, run the `'clean'`, `'undeploy'` and `'deploy'` targets. This will remove all old class files, re-build the application and deploy it. Start up Tomcat again and you should see the following:


SpringApp
http://localhost:8080/springapp/hello.htm

Hello :: SpringApp

Greetings, it is now Sat Nov 10 20:15:32 EST 2007

Products

Lamp *$5.75*

Table *$75.25*

Chair *$22.79*

Done

The updated application

4.5. Adding a form

To provide an interface in the web application to expose the price increase functionality, we add a form that will allow the user to enter a percentage value. This form uses a tag library named `'spring-form.tld'` that is provided with the Spring Framework. We have to copy this file from the Spring distribution (**`'spring-framework-2.5/dist/resources/spring-form.tld'`**) to the **`'springapp/war/WEB-INF/tld'`** directory that we also need to create. Next we must also add a `<taglib/>` entry to the **`'web.xml'`** file.

`'springapp/war/WEB-INF/web.xml'`:

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

<web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

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

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

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

  <jsp-config>
    <taglib>
      <taglib-uri>/spring</taglib-uri>
      <taglib-location>/WEB-INF/tld/spring-form.tld</taglib-location>
    </taglib>
  </jsp-config>

</web-app>
```

We also have to declare this taglib in a page directive in the jsp file, and then start using the tags we have thus imported. Add the JSP page **'priceincrease.jsp'** to the **'war/WEB-INF/jsp'** directory.

'springapp/war/WEB-INF/jsp/priceincrease.jsp':

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<html>
<head>
  <title><fmt:message key="title"/></title>
  <style>
    .error { color: red; }
  </style>
</head>
<body>
<h1><fmt:message key="priceincrease.heading"/></h1>
<form:form method="post" commandName="priceIncrease">
  <table width="95%" bgcolor="f8f8ff" border="0" cellspacing="0" cellpadding="5">
    <tr>
      <td align="right" width="20%">Increase (%):</td>
        <td width="20%">
          <form:input path="percentage"/>
        </td>
        <td width="60%">
          <form:errors path="percentage" cssClass="error"/>
        </td>
    </tr>
  </table>
  <br>
  <input type="submit" align="center" value="Execute">
</form:form>
<a href="<c:url value="hello.htm"/>">Home</a>
</body>
</html>
```

This next class is a very simple JavaBean class, and in our case there is a single property with a getter and setter. This is the object that the form will populate and that our business logic will extract the price increase percentage from.

'springapp/src/springapp/service/PriceIncrease.java':

```
package springapp.service;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class PriceIncrease {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    private int percentage;

    public void setPercentage(int i) {
        percentage = i;
        logger.info("Percentage set to " + i);
    }

    public int getPercentage() {
        return percentage;
    }

}
```

The following validator class gets control after the user presses submit. The values entered in the form will be set on the command object by the framework. The `validate(..)` method is called and the command object (`PriceIncrease`) and a contextual object to hold any errors are passed in.

'springapp/src/springapp/service/PriceIncreaseValidator.java':

```
package springapp.service;

import org.springframework.validation.Validator;
import org.springframework.validation.Errors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class PriceIncreaseValidator implements Validator {
    private int DEFAULT_MIN_PERCENTAGE = 0;
    private int DEFAULT_MAX_PERCENTAGE = 50;
    private int minPercentage = DEFAULT_MIN_PERCENTAGE;
    private int maxPercentage = DEFAULT_MAX_PERCENTAGE;

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    public boolean supports(Class clazz) {
        return PriceIncrease.class.equals(clazz);
    }

    public void validate(Object obj, Errors errors) {
        PriceIncrease pi = (PriceIncrease) obj;
        if (pi == null) {
            errors.rejectValue("percentage", "error.not-specified", null, "Value required.");
        }
        else {
            logger.info("Validating with " + pi + ": " + pi.getPercentage());
            if (pi.getPercentage() > maxPercentage) {
                errors.rejectValue("percentage", "error.too-high",
                    new Object[] {new Integer(maxPercentage)}, "Value too high.");
            }
            if (pi.getPercentage() <= minPercentage) {
                errors.rejectValue("percentage", "error.too-low",
                    new Object[] {new Integer(minPercentage)}, "Value too low.");
            }
        }
    }

    public void setMinPercentage(int i) {
        minPercentage = i;
    }

    public int getMinPercentage() {
        return minPercentage;
    }

    public void setMaxPercentage(int i) {
        maxPercentage = i;
    }

    public int getMaxPercentage() {
        return maxPercentage;
    }

}
```

4.6. Adding a form controller

Now we need to add an entry in the **'springapp-servlet.xml'** file to define the new form and controller. We define objects to inject into properties for `commandClass` and `validator`. We also specify two views, a `formView` that is used for the form and a `successView` that we will go to after successful form processing. The latter can be of two types. It can be a regular view reference that is forwarded to one of our JSP pages. One disadvantage with this approach is, that if the user refreshes the page, the form data is submitted again, and you would end up with a double price increase. An alternative way is to use a redirect, where a response is sent back to the users browser instructing it to redirect to a new URL. The URL we use in this case can't be one of our JSP pages, since they are hidden from direct access. It has to be a URL that is externally reachable. We have chosen to use **'hello.htm'** as my redirect URL. This URL maps to the **'hello.jsp'** page, so this should work nicely.

'springapp/war/WEB-INF/springapp-servlet.xml':

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<!-- the application context definition for the springapp DispatcherServlet -->

<beans>

    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="products">
            <list>
                <ref bean="product1"/>
                <ref bean="product2"/>
                <ref bean="product3"/>
            </list>
        </property>
    </bean>

    <bean id="product1" class="springapp.domain.Product">
        <property name="description" value="Lamp"/>
        <property name="price" value="5.75"/>
    </bean>

    <bean id="product2" class="springapp.domain.Product">
        <property name="description" value="Table"/>
        <property name="price" value="75.25"/>
    </bean>

    <bean id="product3" class="springapp.domain.Product">
        <property name="description" value="Chair"/>
        <property name="price" value="22.79"/>
    </bean>

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
    </bean>

    <bean name="/hello.htm" class="springapp.web.InventoryController">
        <property name="productManager" ref="productManager"/>
    </bean>

    <bean name="/priceincrease.htm" class="springapp.web.PriceIncreaseFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="priceIncrease"/>
        <property name="commandClass" value="springapp.service.PriceIncrease"/>
        <property name="validator">
            <bean class="springapp.service.PriceIncreaseValidator"/>
        </property>
        <property name="formView" value="priceincrease"/>
        <property name="successView" value="hello.htm"/>
        <property name="productManager" ref="productManager"/>
    </bean>
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

Next, let's take a look at the controller for this form. The `onSubmit(..)` method gets control and does some logging before it calls the `increasePrice(..)` method on the `ProductManager` object. It then returns a `ModelAndView` passing in a new instance of a `RedirectView` created using the URL for the success view.

'springapp/src/web/PriceIncreaseFormController.java':

```
package springapp.web;

import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.view.RedirectView;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import springapp.service.ProductManager;
import springapp.service.PriceIncrease;

public class PriceIncreaseFormController extends SimpleFormController {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    private ProductManager productManager;

    public ModelAndView onSubmit(Object command)
            throws ServletException {

        int increase = ((PriceIncrease) command).getPercentage();
        logger.info("Increasing prices by " + increase + "%.");

        productManager.increasePrice(increase);

        logger.info("returning from PriceIncreaseForm view to " + getSuccessView());

        return new ModelAndView(new RedirectView(getSuccessView()));
    }

    protected Object formBackingObject(HttpServletRequest request) throws ServletException {
        PriceIncrease priceIncrease = new PriceIncrease();
        priceIncrease.setPercentage(20);
        return priceIncrease;
    }

    public void setProductManager(ProductManager productManager) {
        this.productManager = productManager;
    }

    public ProductManager getProductManager() {
        return productManager;
    }

}
```

We are also adding some messages to the **'messages.properties'** resource file.

'springapp/war/WEB-INF/classes/messages.properties':

```
title=SpringApp
heading=Hello :: SpringApp
```

```
greeting=Greetings, it is now
priceincrease.heading=Price Increase :: SpringApp
error.not-specified=Percentage not specified!!!
error.too-low=You have to specify a percentage higher than {0}!
error.too-high=Don''t be greedy - you can''t raise prices by more than {0}%!
required=Entry required.
typeMismatch=Invalid data.
typeMismatch.percentage=That is not a number!!!
```

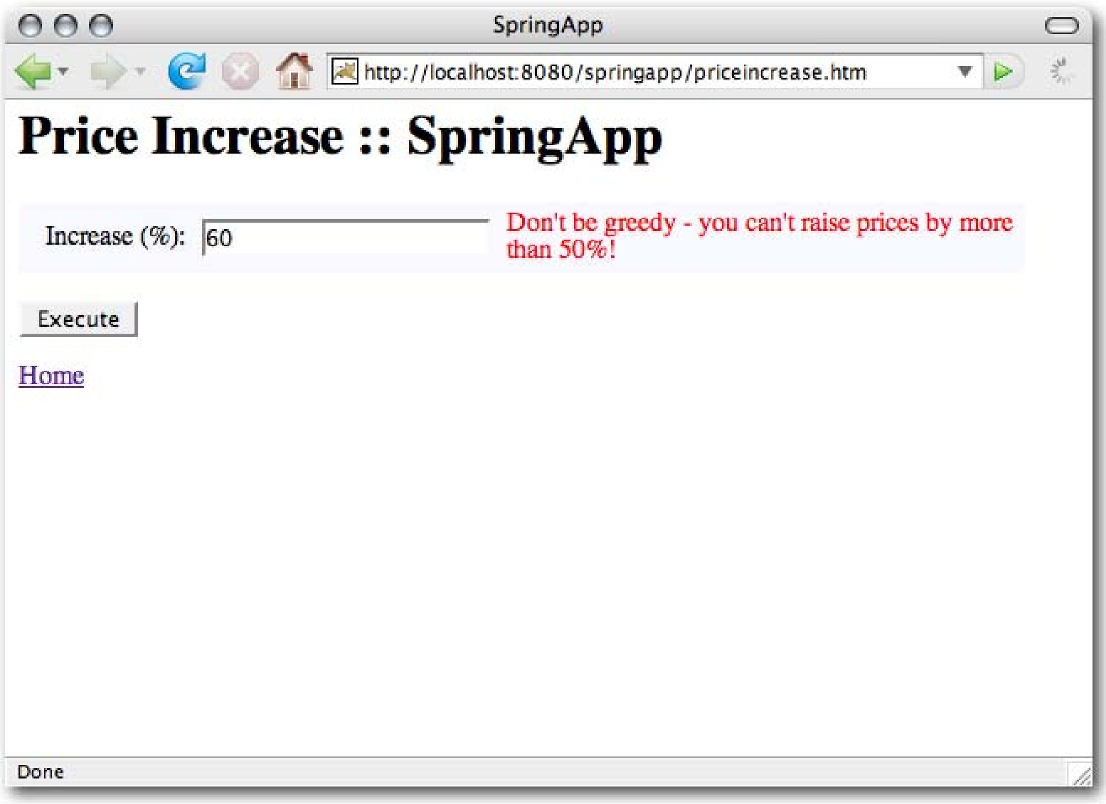
Compile and deploy all this and after reloading the application we can test it. This is what the form looks like with errors displayed.

Finally, we will add a link to the price increase page from the **'hello.jsp'**.

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
  <head><title><fmt:message key="title"/></title></head>
  <body>
    <h1><fmt:message key="heading"/></h1>
    <p><fmt:message key="greeting"/> <c:out value="${model.now}"/></p>
    <h3>Products</h3>
    <c:forEach items="${model.products}" var="prod">
      <c:out value="${prod.description}"/> <i>$<c:out value="${prod.price}"/></i><br><br>
    </c:forEach>
    <br>
    <a href="<c:url value="priceincrease.htm"/>">Increase Prices</a>
    <br>
  </body>
</html>
```

Now, run the 'deploy' and 'reload' targets and try the new price increase feature.


SpringApp

http://localhost:8080/springapp/priceincrease.htm

Price Increase :: SpringApp

Increase (%): 60 Don't be greedy - you can't raise prices by more than 50%!

The updated application

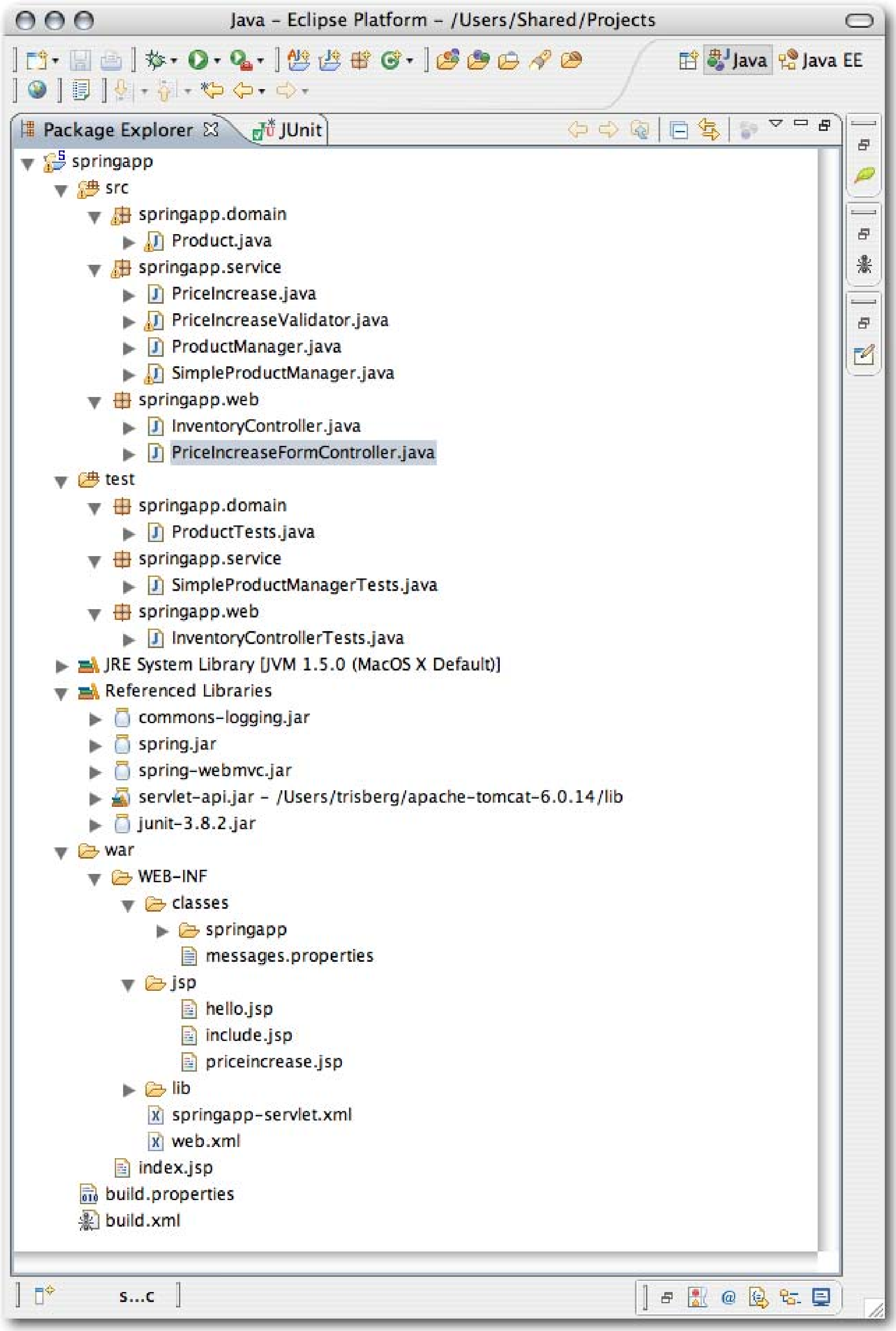
4.7. Summary

Let's look at what we did in Part 4.

- We renamed our controller to `InventoryController` and gave it a reference to a `ProductManager` so we could retrieve a list of products to display.

- Next we modified the JSP page to use a message bundle for static text and also added a forEach loop to show the dynamic list of products.
- Then we defined some test data to populate business objects we modified the JSP page to use a message bundle for static text and also added a forEach loop to show the dynamic list of products.
- Next we modified the JSP page to use a message bundle for static text and also added a forEach loop to show the dynamic list of products.
- After this worked we created a form to provide the ability to increase the prices. Next we modified the JSP page to use a message bundle for static text and also added a forEach loop to show the dynamic list of products.
- Finally we created the form controller and a validator and deployed and tested the new features.

Find below a screen shot of what your project directory structure must look like after following the above instructions.



```
springapp
  src
    springapp.domain
      Product.java
    springapp.service
      PriceIncrease.java
      PriceIncreaseValidator.java
      ProductManager.java
      SimpleProductManager.java
    springapp.web
      InventoryController.java
      PriceIncreaseFormController.java
  test
    springapp.domain
      ProductTests.java
    springapp.service
      SimpleProductManagerTests.java
    springapp.web
      InventoryControllerTests.java
  JRE System Library [JVM 1.5.0 (MacOS X Default)]
  Referenced Libraries
    commons-logging.jar
    spring.jar
    spring-webmvc.jar
    servlet-api.jar - /Users/trisberg/apache-tomcat-6.0.14/lib
    junit-3.8.2.jar
  war
    WEB-INF
      classes
        springapp
        messages.properties
      jsp
        hello.jsp
        include.jsp
        priceincrease.jsp
      lib
      springapp-servlet.xml
      web.xml
    index.jsp
  build.properties
  build.xml
```

Prev | Home | Next
Chapter 3. Developing the Business Logic | Sponsored by SpringSource | Chapter 5. Implementing Database Persistence