Developing a Spring Framework MVC application step-by-step

This document is created by Yu Zhao and edited by Chunfeng Liu.

Table of Contents

Chapter 0: Things You Need to Know

We recommend using your personal computer to complete the practice in tutorials.

Integrated development environment (IDE):

Download recommended IDE Spring Tool Suite (STS) from **https://spring.io/tools/sts/legacy**
Download the version 3.9.4 based on your OS of personal computer.

Note: Please make sure you download the right version. All the contents in this document are created based on the version of 3.9.*

Download JDK

The latest version (3.9.4) of STS only supports version 8 or higher version of JDK. If your JDK version is lower than 8, make sure you update your JDK to 8 or higher version. You can download JDK 8 from this page (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html). Please select the correct JDK based on your OS.

Code:

The final source codes of this project is available at: https://bitbucket.org/ianliu0420/springapp-2016

* In this document, all screenshots have the URL http://localhost:8080/XXX/XXX, which is my own configuration of Tomcat. You need to use your own Tomcat (or other server) port number. (Default is 8080)

Proxy:

If you are using your own laptop, please skip the following section, and continue the tutorial from Chapter 1. If you are working on the lab computer, you need to configure the proxy settings of Maven as follow to make everything work fine.

Please go through the document of "elec5619-maven.doc" to set up the proxy and install maven into your STS.

If the above proxy instruction does not work for the computer, please try to use Maven in command line.

Steps to take configure the computers:

1. Unzip the distribution archive, i.e. *apache-maven-3.1.0-bin.zip* to the directory you wish to install Maven 3.1.0 (**in our case, unzip the file to the Desktop**). These instructions assume you chose *C:\Program Files\Apache Software Foundation*. The subdirectory apache-maven-3.1.0 will be created from the archive.
2. Add the **M2_HOME** environment variable by opening up the system properties (WinKey + Pause), selecting the "Advanced" tab, and the "Environment Variables" button, then adding the *M2_HOME* variable in the user variables with the value C:\Program Files\Apache Software Foundation\apache-maven-3.1.0. Be sure to omit any quotation marks around the path even if it contains spaces. **Note**:

For Maven 2.0.9, also be sure that the M2_HOME doesn't have a '\' as last character.

3. In the same dialog, add the M2 environment variable in the user variables with the value **%M2_HOME%\bin**.
4. **Optional**: In the same dialog, add the MAVEN_OPTS environment variable in the user variables to specify JVM properties, e.g. the value -Xms256m -Xmx512m. This environment variable can be used to supply extra options to Maven.
5. In the same dialog, update/create the Path environment variable in the user variables and prepend the value **%M2%** to add Maven available in the command line.
6. In the same dialog, make sure that **JAVA_HOME** exists in your user variables or in the system variables and it is set to the location of your JDK, e.g.C:\Program Files\Java\jdk1.5.0_02 and that **%JAVA_HOME%\bin** is in your Path environment variable.
7. Open a *new* command prompt (Winkey + R then type cmd) and run **mvn --version** to verify that it is correctly installed.

Useful Maven Commands:

ref: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

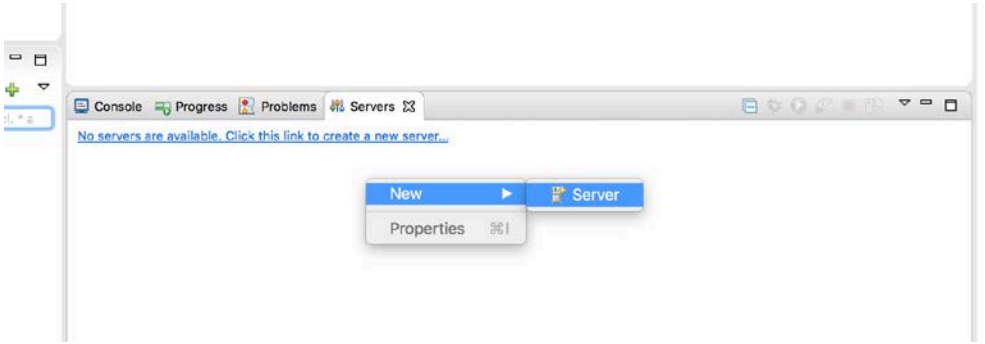
1. *mvn --version* ⇒ check the version of the Maven installation
2. *mvn test* ⇒ run tests within the test package
3. *mvn install* or *mvn compile* ⇒ build the project
4. *mvn clean* ⇒ clean the project that has been built, remove the binary files
5. *mvn tomcat:run* ⇒ run project on built-in Tomcat server
6. *mvn -Dmaven.tomcat.port=8181 tomcat:run-war* ⇒ run your application on Tomcat on port 8181.

Chapter 1: Basic Application and Environment Setup

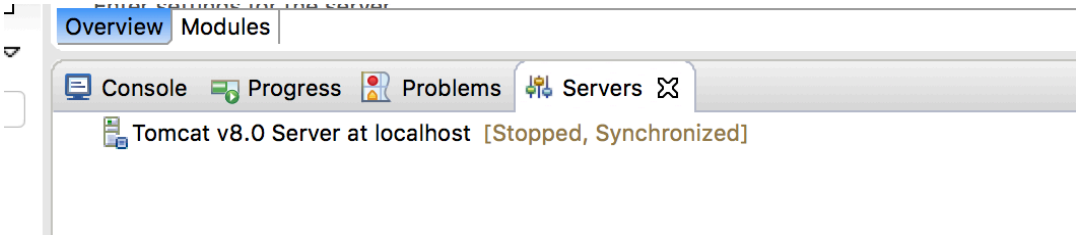
1.1 Install and configure tomcat server on STS

You can follow this video (https://www.youtube.com/watch?v=n14rpj_08wM) to complete this step.

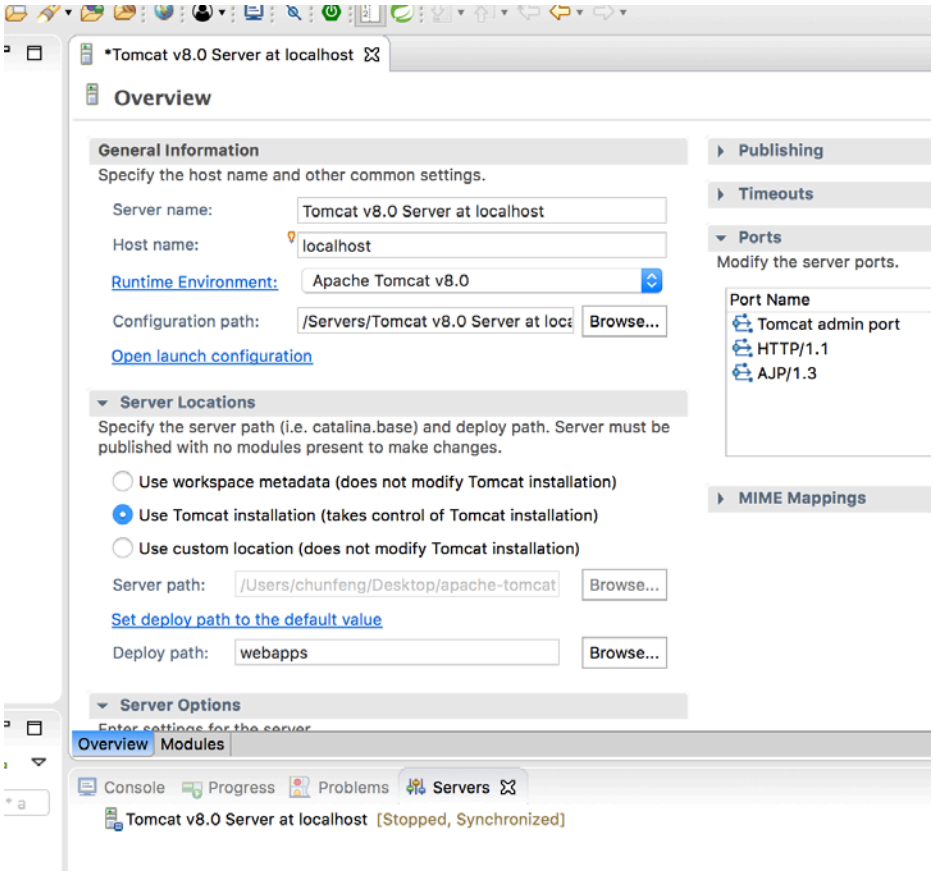
1. Download the Tomcat 8.0.53 server from https://tomcat.apache.org/download-80.cgi
2. Then, decompress Tomcat into a folder, such as "C:\Elec5619\"
3. On STS, Select Windows->Show view->Servers
4. Under "Servers", right click, and select "new", than select "Server"



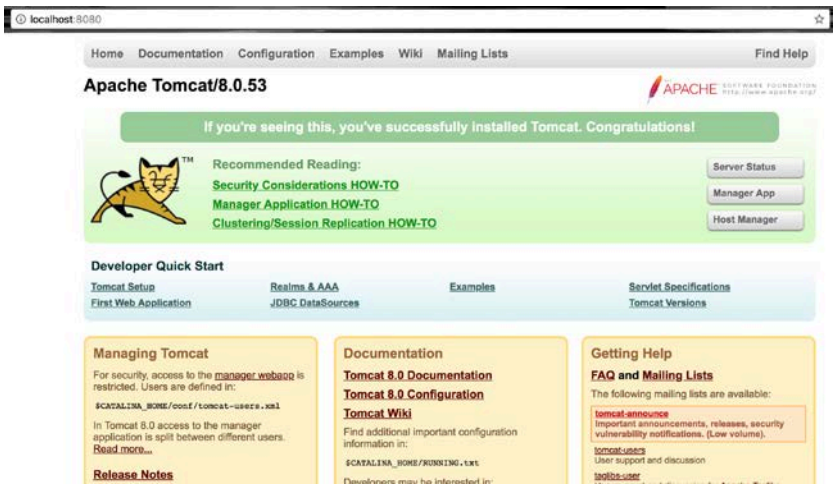
5. In the pop-up window, select "Apache->Tomcat v8.0 Server", click "next…"
6. Then another pop-up window will be displayed.
7. In the pop-up window, paste the root directory of your Tomcat 8, for example "C:\Elec5619\apache-tomcat-8.0.53" in the "Tomcat installation directory" space, and click Finish.
8. Then your "Servers" Section will be like this:



9. Right click the server, and select open. In the new window, in "Server Location" section, select "Use Tomcat installation". In the "Deploy path", please type "webapps". Then save the settings by "Ctrl+s".



10. By now, you have already set up a Tomcat 8 server on your STS.
11. In this step, we will test whether the server is able to run correctly.
    1) Right click the server
    2) Select “Start”
    3) Open your web browser, type the link http://localhost:8080/
    4) If your web page like following, that means you have already correctly configured a web server on your machine. Congrats.



1.2 Create a basic SpringMVC project

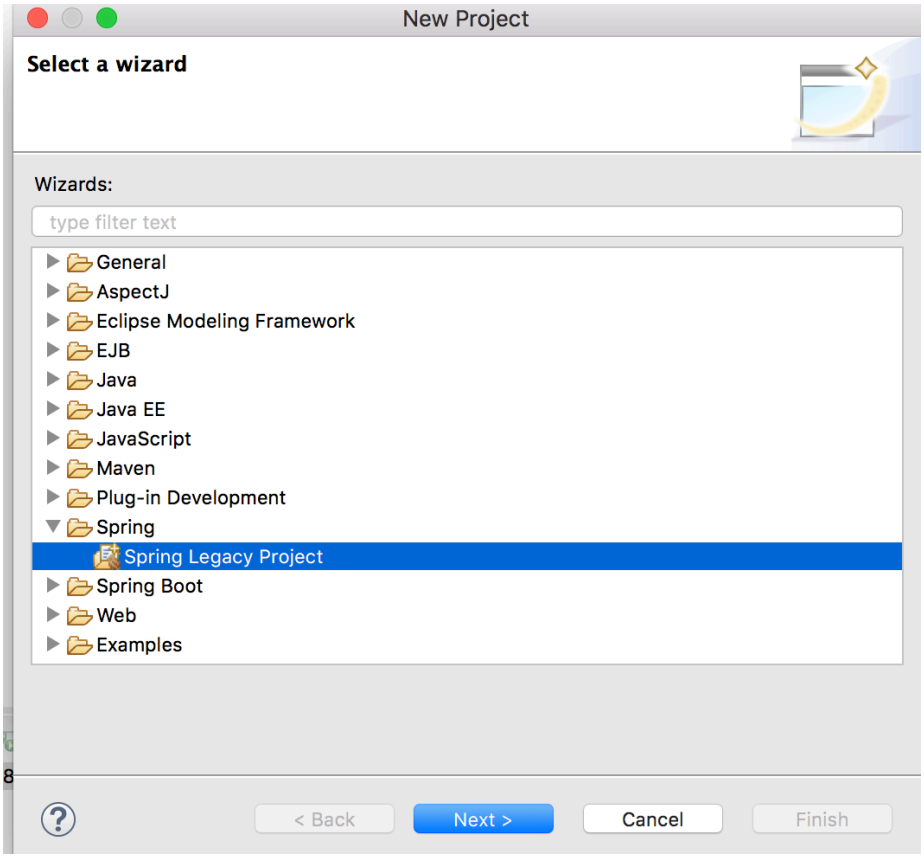
Now, let's create a basic Spring MVC project, and it will be used as the base of further tutorials in this document.

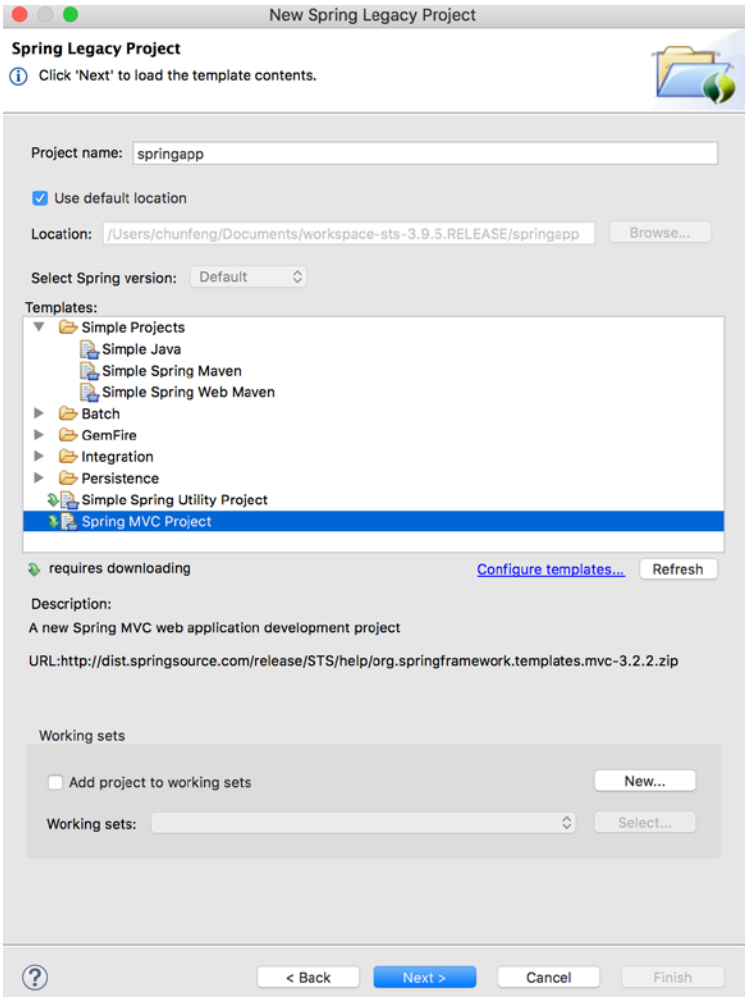
Steps to create the project (in STS):

File -> New -> Project -> Spring -> Spring Legacy Project (click next) -> Spring MVC Project

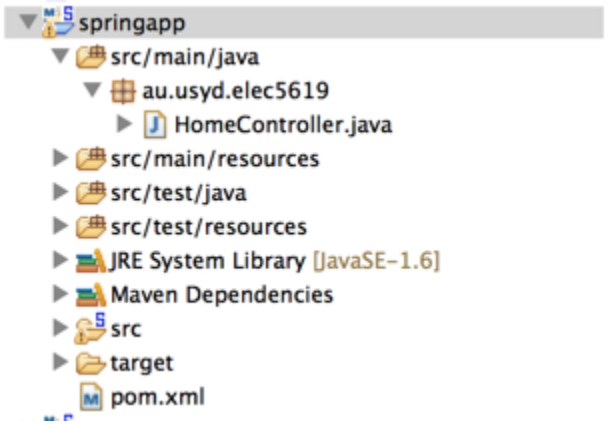
Project Name: springapp

Top level package: au.usyd.elec5619



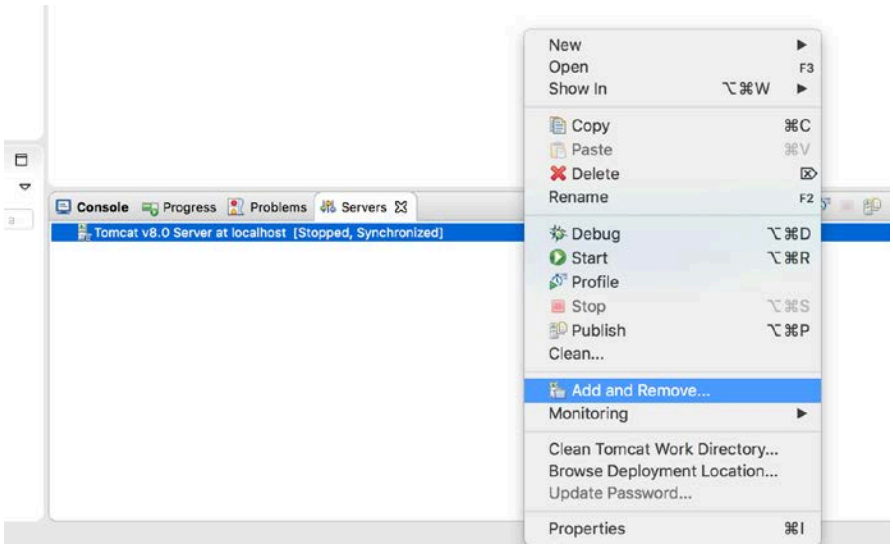


In the package explorer, you will get a project with the following structure.

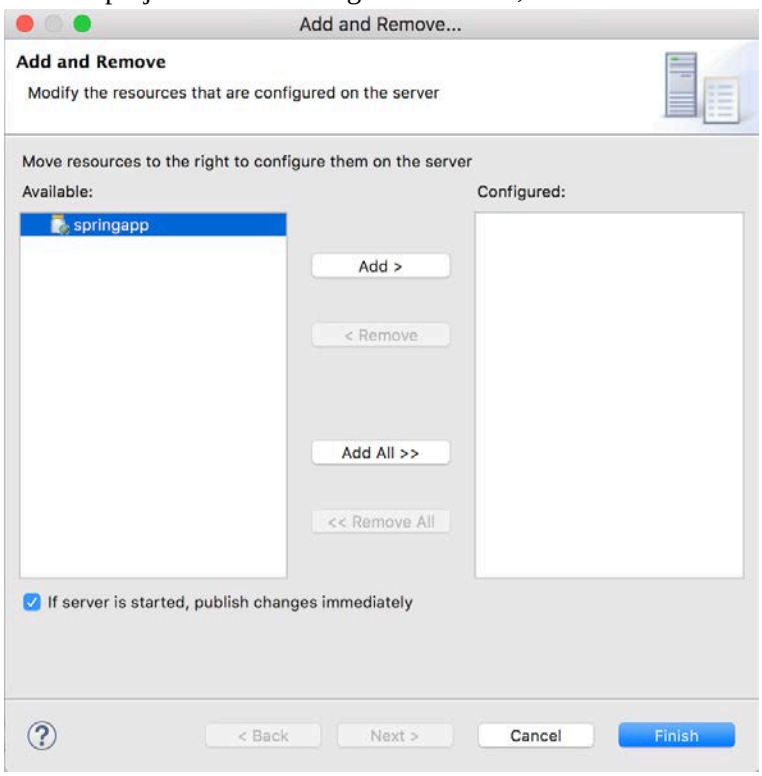


Now we need to deploy the project into the Tomcat server:

1. Right click Tomcat server, then select “Add and Remove”



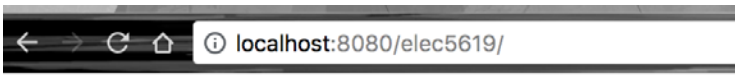
2. Add the project to the “Configured” section, then click finish



3. Then right click Tomcat server, then click “Start”

Now you have deployed your project to the Tomcat server, and you can access the pape from the URL: http://localhost:8080/elec5619/

And the page will be shown like this:



Hello world!

The time on the server is August 8, 2018 3:10:14 PM AEST.

You can check the detail procedures from this video (https://www.youtube.com/watch?v=S5cbm6SDyvU), and the procedures in the video might be different because of the different version of STS.

Chapter 2: Developing and Configuring the Views and the Controller

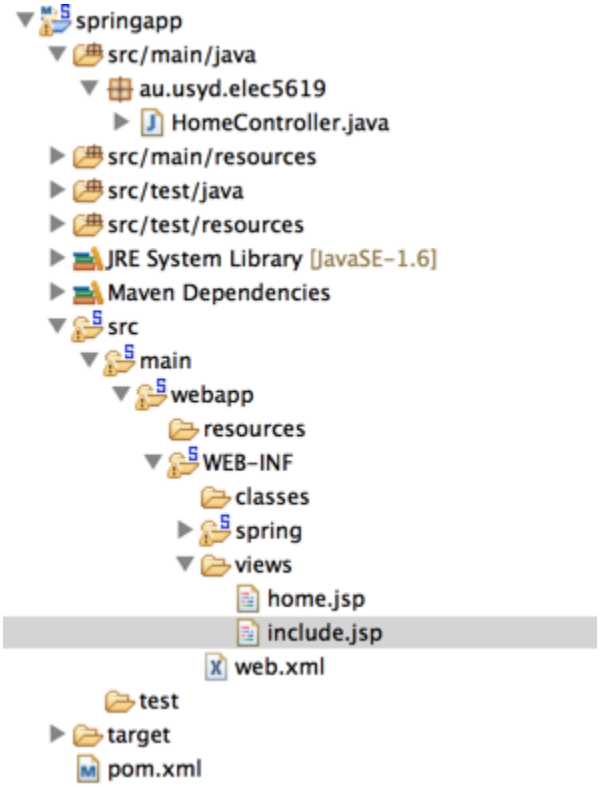
2.1 Configure JSTL and add JSP header file

We will be creating a 'header' file that will be included in every JSP page that we're going to write. We ensure the same definitions are included in all our JSPs simply by including the header file. We're also going to put all JSPs in a directory named '**views**' under the '**WEB-INF**' directory. This will ensure that views can only be accessed via the controller since it will not be possible to access these pages directly via a URL. This strategy might not work in some application servers and if this is the case with the one you are using, move the '**views**' directory up a level. You would then use '**springapp/war/views**' as the directory instead of '**springapp/war/WEB-INF/views**' in all the code examples that will follow.

2.1.1 Create the include.jsp file

First we create the header file for inclusion in all the JSPs we create.
In the file '**springapp/src/main/webapp/WEB-INF/views/include.jsp**':

```
<%@ page session="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```



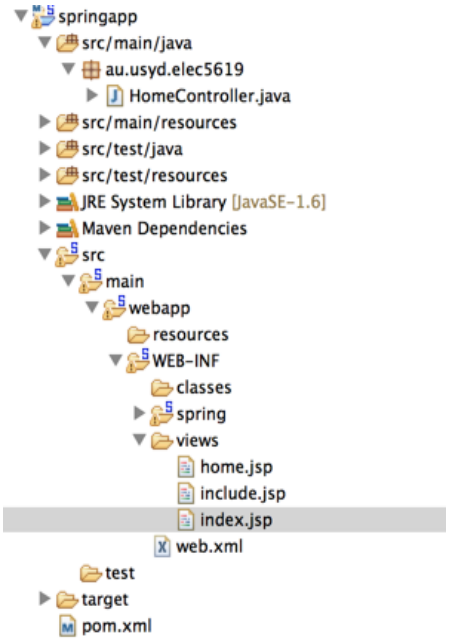
2.1.2 Create the index.jsp file

Now we can update '**index.jsp**' to use this include file and since we are using JSTL, we can use the <c:redirect/> tag for redirecting to our front Controller. This means all requests for '**index.jsp**' will go

through our application framework. Just delete the current contents of 'index.jsp' and replace it with the following:

In the file **‘springapp/src/main/webapp/WEB-INF/views/index.jsp’**

```
<%@ include file="/WEB-INF/views/include.jsp" %>
<%-- Redirected because we can't set the welcome page to a virtual URL. --%>
<c:redirect url="/hello.htm"/>
```



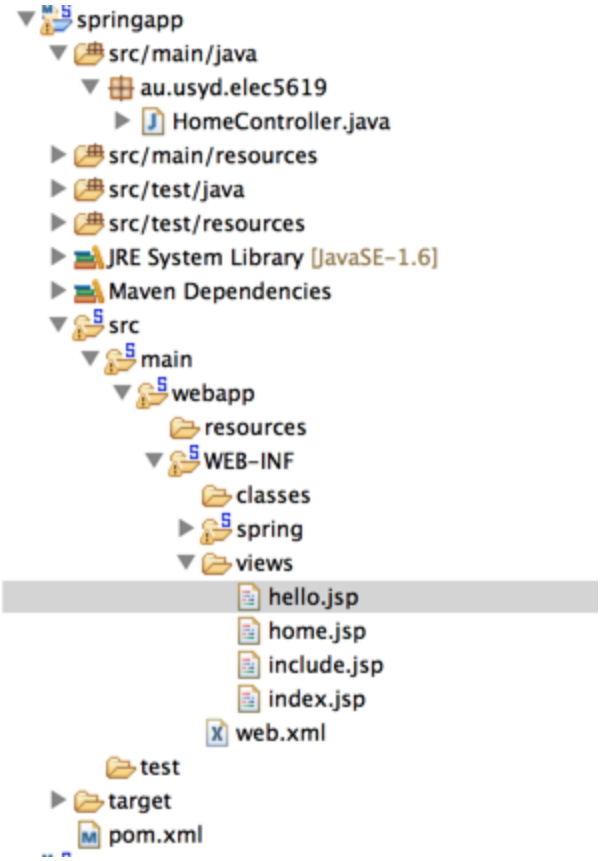
2.1.3 Create the hello.jsp file

Now create another file called ‘**hello.jsp**’ inside the ‘**views**’ folder with the following content.

In the file **‘springapp/src/main/webapp/WEB-INF/views/hello.jsp’**

```
<%@ include file="/WEB-INF/views/include.jsp" %>
<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
    <p>Greetings, it is now <c:out value="${now}"/></p>
  </body>
</html>
```

And after creating the ‘**hello.jsp**’ file the folder structure will be similar to the image below.



2.2 Create a controller for the view

So far, you've create all the JSP files, which we call the "**views**", for this application. If you want to access those views through a browser, you need to provide other Java classes, which we call "**controllers**".

Now create a new Java file called "**HelloController**" in "**springapp/src/main/java/au.usyd.elec5619**", with the following content.

```
package au.usyd.elec5619;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class HelloController implements Controller {
        protected final Log logger = LogFactory.getLog(getClass());
        @Override
```

```
        public ModelAndView handleRequest(HttpServletRequest arg0,
                        HttpServletResponse arg1) throws Exception {

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

       return new ModelAndView("hello", "now", now);
        }

}
```

And add a new line in "**springapp/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml**".

```
<beans:bean name="/hello.htm" class="au.usyd.elec5619.HelloController"/>
```

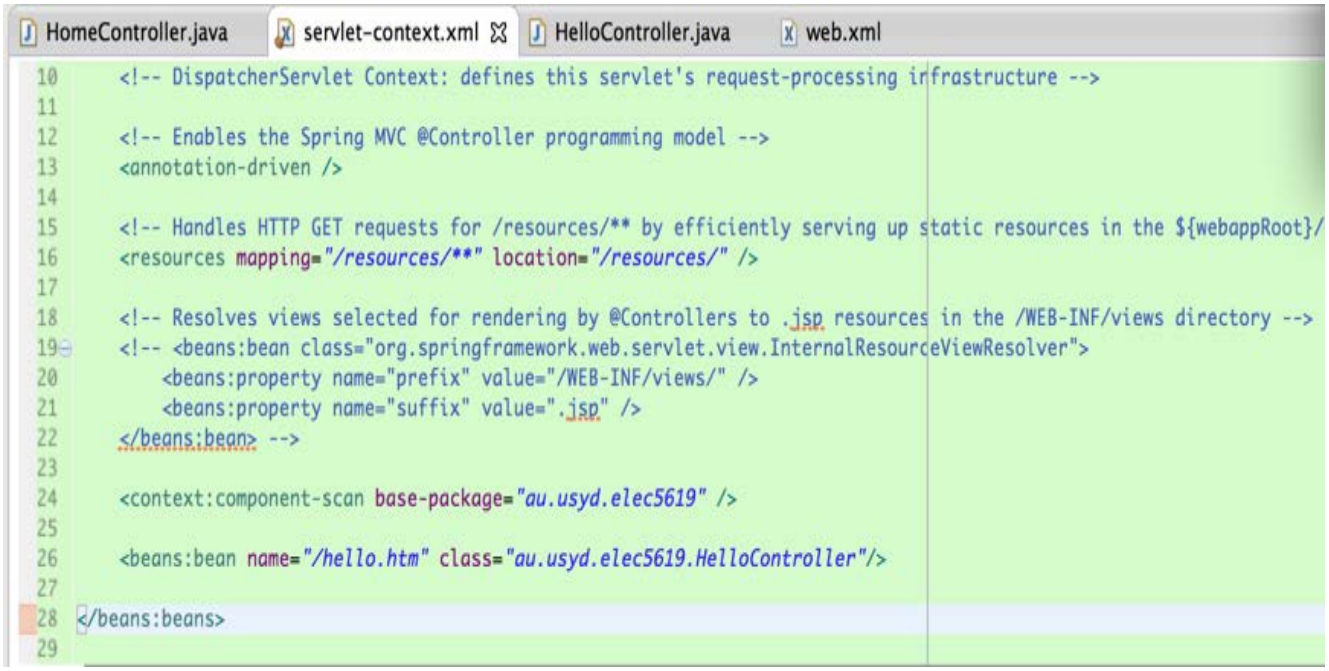
Now if you run this application on a server, and go to the URL: http://localhost:8080/elec5619/hello.htm
You will be able to see the "**hello.jsp**" page

2.3 Decouple the view from the controller

Now please open "**springapp/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml**" and "**springapp/src/main/java/au.usyd/elec5619/HelloController.java**".
In "**servlet-context.xml**", please comment out the following lines (should be at around line 19 to 22):

```
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
```

```
        <beans:property name="suffix" value=".jsp" />
</beans:bean>
```



Then run this application on server and go to http://localhost:8080/elec5619/hello.htm again and see what happens.

It’s a 404, which means your application cannot find the specified resource. And please have a look at the output in the Eclipse or STS console.

```
Aug 12, 2013 1:44:22 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-9080"]
Aug 12, 2013 1:44:22 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Aug 12, 2013 1:44:22 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1627 ms
INFO : au.usyd.elec5619.HelloController - Returning hello view with Mon Aug 12 13:44:25 EST 2013
WARN : org.springframework.web.servlet.PageNotFound - No mapping found for HTTP request with URI [/elec5619/hello] in DispatcherServlet with name 'appServlet'
```

Now, please navigate to the "**HelloController.java**" tab opening in your IDE, and comment out the line, which should be the return statement:

```
return new ModelAndView("hello", "now", now);
```

And copy and paste "return new ModelAndView("WEB-INF/views/hello.jsp", "now", now);" as the return statement of the handleRequest() function.

Again, restart the server and go to http://localhost:8080/elec5619/hello.htm and see what happens.

The lines we commented out in "**servlet-context.xml**" are used for mapping to the views. Now, what we need to provide as the first parameter in the highlighted statement of the above image is the entire path to the view page, which is the "**hello.jsp**" page. And to simplify developers' work and using logical name to map to different view pages, we need to use the "**InternalResourceViewResolver**" and specify the "**prefix**" and "**suffix**" for the logical name you provide.

Now you can leave your code as is (and you need to write the whole path to views again and again) or you can uncomment the 2 commented sections and remove the lines we just added.

Chapter 3: Developing the Business Logic

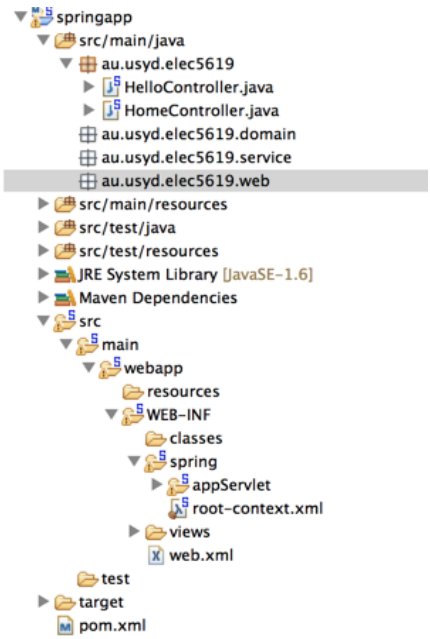
This is Part 3 of a step-by-step tutorial on how to develop a Spring application. In this section, we will adopt a pragmatic Test-Driven Development (TDD) approach for creating the domain objects and implementing the business logic for our inventory management system. This means we'll "code a little, test a little, code some more then test some more". In Part 1 we configured the environment and set up a basic application. In Part 2 we refined the application by decoupling the view from the controller.

Spring is about making simple things easy and the hard things possible. The fundamental construct that makes this possible is Spring's use of Plain Old Java Objects (POJOs). POJOs are essentially plain old Java classes free from any contract usually enforced by a framework or component architecture through subclassing or the implementation of interfaces. POJOs are plain old objects that are free from such constraints, making object-oriented programming possible once again. When you are working with Spring, the domain objects and services you implement will be POJOs. In fact, almost everything you implement should be a POJO. If it's not, you should be sure to ask yourself why that is. In this section, we will begin to see the simplicity and power of Spring.

3.0 Refine packages

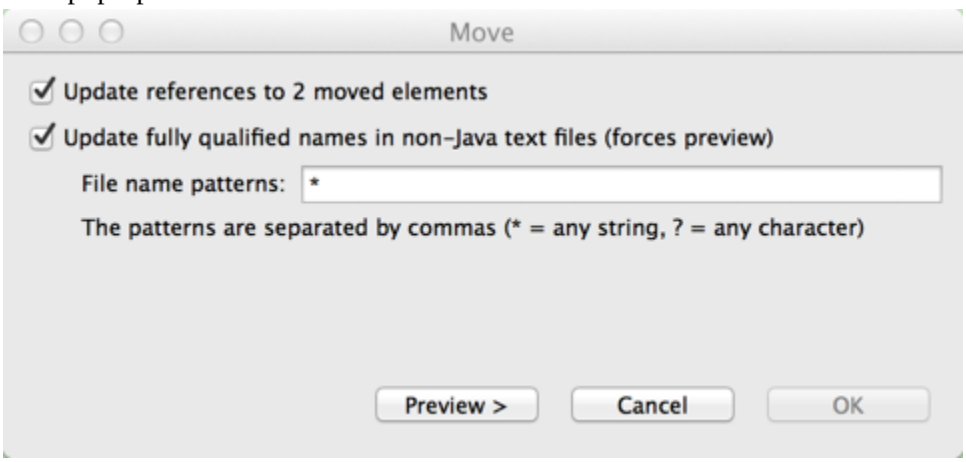
So far, we are putting all our Java code directly in the "**au.usyd.elec5619**" package, which is ok for small project. However, it is not a good practice to do so. We need to put our "M"s, "V"s, and "C"s in different packages.

Now please create 3 sub-packages under the "**au.usyd.elec5619**" package, with the name "domain", "web", and "service".

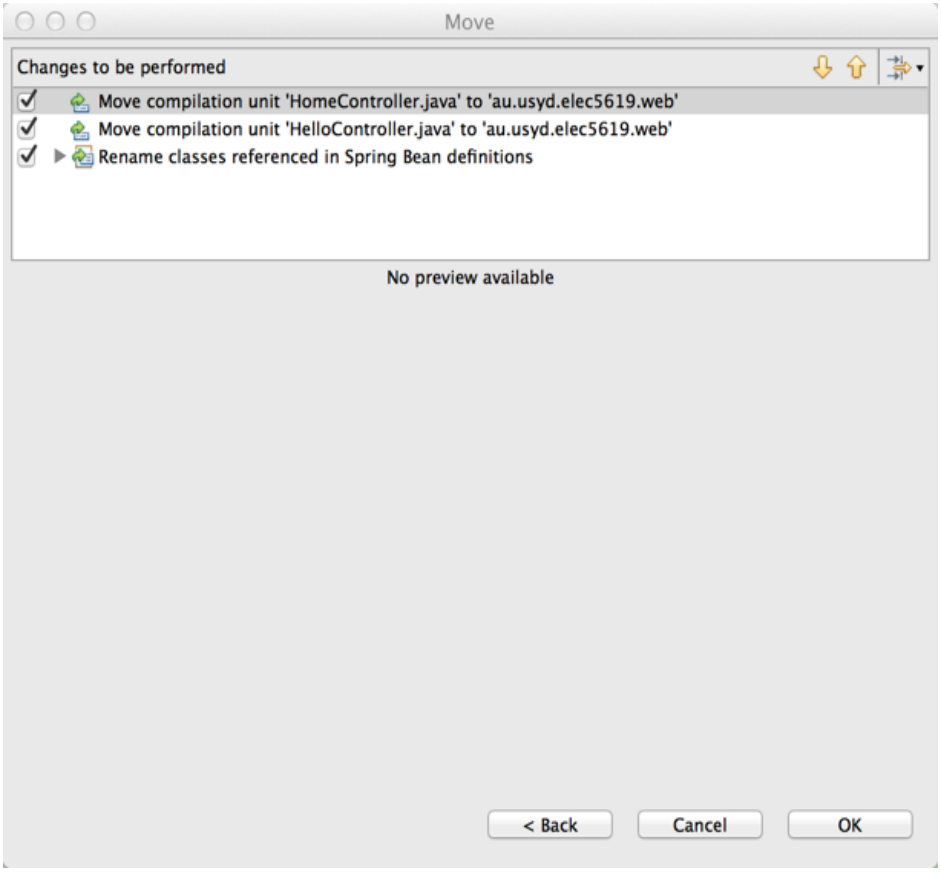


And now, you can drag and drop the 2 controllers into the "**au.usyd.elec5619.web**" package. And you will

see a pop-up window like:



Select "**Update fully qualified names in non-java text files (forces preview)**", and click "**Preview**". And you will see a second pop-up screen similar to the image below.



And click "**OK**".

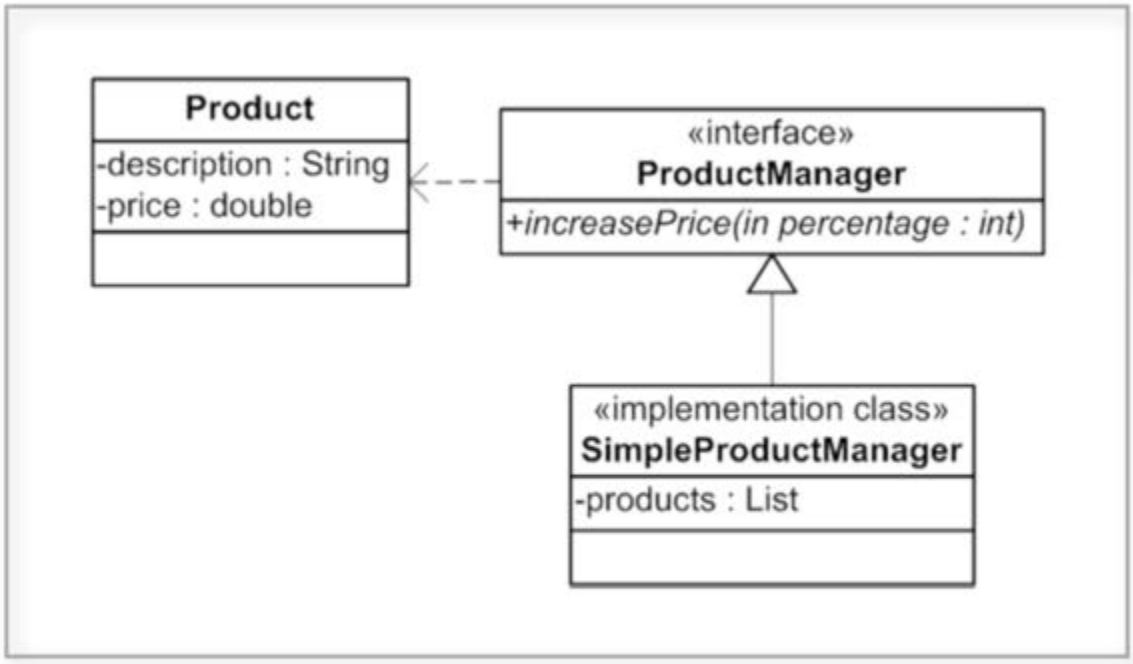
Then rerun your server and test if all necessary mappings have been correctly updated. (Everything should be working exactly as what you have been doing.)

3.1 Review the business case of the Inventory Management System

In our inventory management system, we have the concept of a product and a service for handling them. In particular, the business has requested the ability to increase prices across all products. Any decrease will be done on an individual product basis, but this feature is outside the scope of our application. The validation rules for price increase are:

- The maximum increase is limited to 50%.
- The minimum increase must be greater than 0%.

Find below a class diagram of our inventory management system.



The class diagram for the inventory management system

3.2 Add some classes for business logic

Let's now add some business logic in the form of a Product class and a service called ProductManager service that will manage all the products.

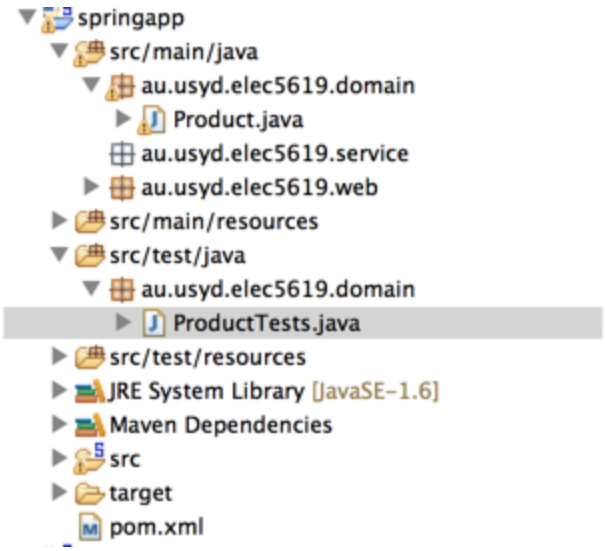
First we implement the Product class as a POJO with a default constructor (automatically provided if we don't specify any constructors) and getters and setters for its properties 'description' and 'price'. Let's also make it Serializable, not necessary for our application, but could come in handy later on when we persist and store its state. The class is a domain object, so it belongs in the 'domain' package.

‘**springapp/src/main/java/au.usyd.elec5619.domain/Product.java**’:

Product.java

```java
package au.usyd.elec5619.domain;

import java.io.Serializable;

public class Product implements Serializable {

    private String description;
    private Double price;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Description: " + description + ";");
        buffer.append("Price: " + price);
        return buffer.toString();
    }
}

```

Now we write the unit tests for our Product class. Some developers don't bother writing tests for getters and setters or so-called 'auto-generated' code. It usually takes much longer to engage in the debate (as this paragraph demonstrates) on whether or not getters and setters need to be unit tested as they're so 'trivial'. We write them because: a) they are trivial to write; b) having the tests pays dividends in terms of the time saved for the one time out of a hundred you may be caught out by a dodgy getter or setter; and c) they improve test coverage. We create a Product stub and test each getter and setter as a pair in a single test. Usually, you will write one or more test methods per class method, with each test method testing a particular condition in a class method such as checking for a null value of an argument passed into the method.



In the file '**springapp/src/test/java/au.usyd.elec5619.domain/ProductTest.java'**

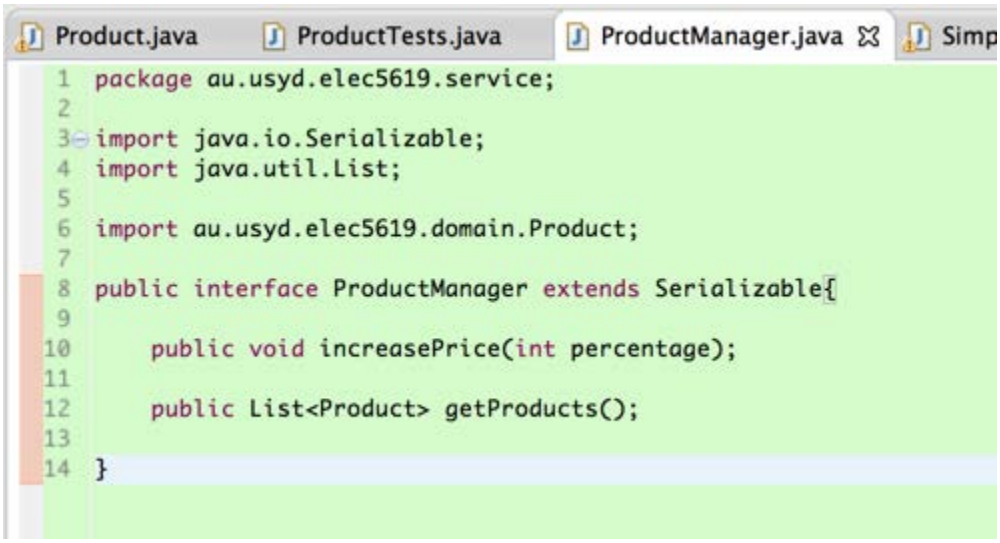
Product.java ProductTest.java

```
1  package au.usyd.elec5619.domain;
2
3  import junit.framework.TestCase;
4
5  public class ProductTest extends TestCase {
6
7      private Product product;
8
9      protected void setUp() throws Exception {
10         product = new Product();
11     }
12
13     public void testSetAndGetDescription() {
14         String testDescription = "aDescription";
15         assertNull(product.getDescription());
16         product.setDescription(testDescription);
17         assertEquals(testDescription, product.getDescription());
18     }
19
20     public void testSetAndGetPrice() {
21         double testPrice = 100.00;
22         assertEquals(0, 0, 0);
23         product.setPrice(testPrice);
24         assertEquals(testPrice, product.getPrice(), 0);
25     }
26
27 }
```

Next we create the ProductManager. This is the service responsible for handling products. It contains two methods: a business method increasePrice() that increases prices for all products and a getter method

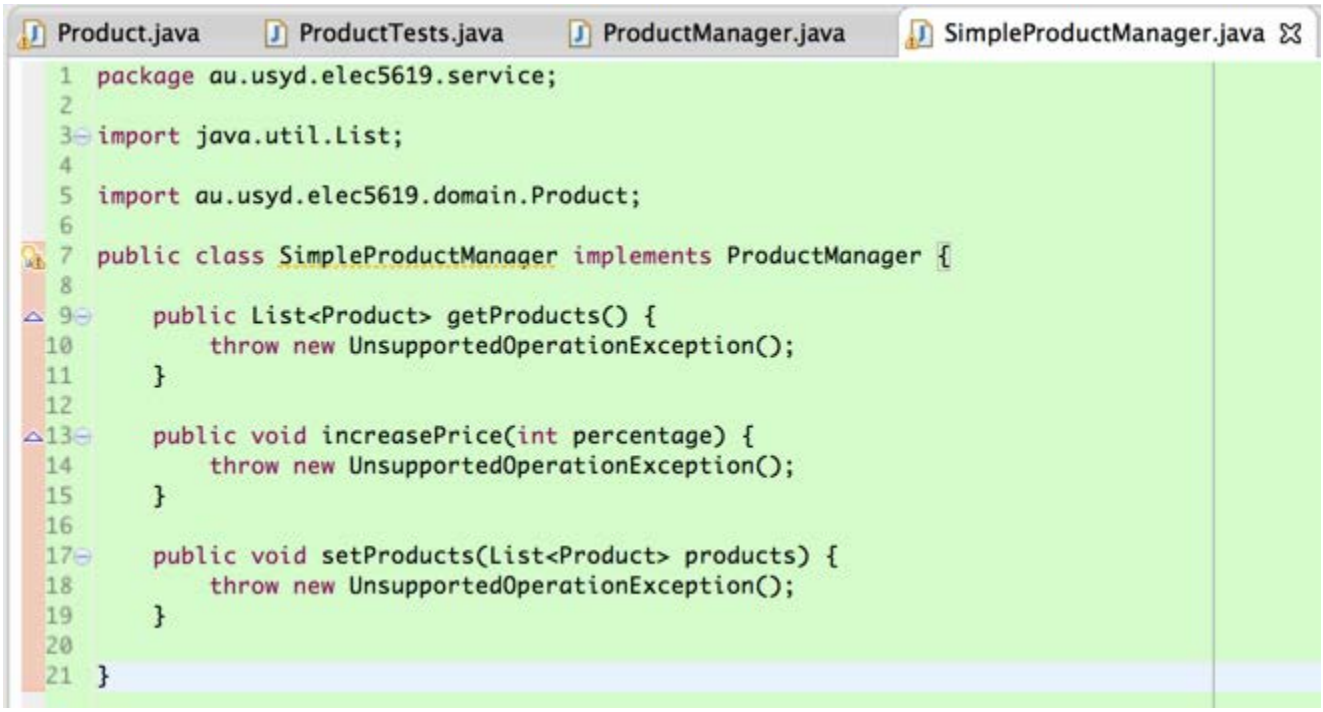
getProducts() for retrieving all products. We have chosen to make it an interface instead of a concrete class for an number of reasons. First of all, it makes writing unit tests for Controllers easier (as we'll see in the next chapter). Secondly, the use of interfaces means JDK proxying (a Java language feature) can be used to make the service transactional instead of CGLIB (a code generation library).

In the file '**springapp/src/main/java/au.usyd.elec5619.service/ProductManager.java'**



Let's create the SimpleProductManager class that implements the ProductManager interface.

In the file '**springapp/src/main/java/au.usyd.elec5619.service/SimpleProductManager.java'**



Before we implement the methods in SimpleProductManager, we're going to define some tests first. The

strictest definition of Test Driven Development (TDD) is to always write the tests first, then the code. A looser interpretation of it is more akin to Test Oriented Development (TOD), where we alternate between writing code and tests as part of the development process. The most important thing is for a codebase to have as complete a set of unit tests as possible, so how you achieve it becomes somewhat academic. Most TDD developers, however, do agree that the quality of tests is always higher when they are written at around the same time as the code that is being developed, so that's the approach we're going to take.

To write effective tests, you have to consider all the possible pre- and post-conditions of a method being tested as well as what happens within the method. Let's start by testing a call to getProducts() returns null.

In the file '**springapp/src/test/java/au.usyd.elec5619.service/SimpleProductManagerTest.java'**

SimpleProductManagerTest.java

```
1  package au.usyd.elec5619.service;
2
3  import junit.framework.TestCase;
4
5  public class SimpleProductManagerTest extends TestCase {
6
7      private SimpleProductManager productManager;
8
9      protected void setUp() throws Exception {
10         productManager = new SimpleProductManager();
11     }
12
13     public void testGetProductsWithNoProducts() {
14         productManager = new SimpleProductManager();
15         assertNull(productManager.getProducts());
16     }
17
18 }
```

Rerun all the Maven tests target and the test should fail as getProducts() has yet to be implemented. It's usually a good idea to mark unimplemented methods by getting them to throw an UnsupportedOperationException.

Next we implement a test for retrieving a list of stub products populated with test data. We know that we'll need to populate the products list in the majority of our test methods in SimpleProductManagerTests, so we define the stub list in JUnit's setUp(), a method that is invoked before each test method is called.

In the file '**springapp/src/test/java/au.usyd.elec5619.service/SimpleProductManagerTest.java'**

SimpleProductManagerTest.java

```java
package au.usyd.elec5619.service;

import java.util.ArrayList;
import java.util.List;

import junit.framework.TestCase;
import au.usyd.elec5619.domain.Product;

public class SimpleProductManagerTest extends TestCase {

    private SimpleProductManager productManager;
    private List<Product> products;

    private static int PRODUCT_COUNT = 2;

    private static Double CHAIR_PRICE = new Double(20.50);
    private static String CHAIR_DESCRIPTION = "Chair";

    private static String TABLE_DESCRIPTION = "Table";
    private static Double TABLE_PRICE = new Double(150.10);

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
        products = new ArrayList<Product>();

        // stub up a list of products
        Product product = new Product();
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

        product = new Product();
        product.setDescription("Table");
        product.setPrice(TABLE_PRICE);
        products.add(product);

        productManager.setProducts(products);
    }
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        assertNull(productManager.getProducts());
    }

    public void testGetProducts() {
        List<Product> products = productManager.getProducts();
        assertNotNull(products);
        assertEquals(PRODUCT_COUNT, productManager.getProducts().size());

        Product product = products.get(0);
        assertEquals(CHAIR_DESCRIPTION, product.getDescription());
        assertEquals(CHAIR_PRICE, product.getPrice());

        product = products.get(1);
        assertEquals(TABLE_DESCRIPTION, product.getDescription());
        assertEquals(TABLE_PRICE, product.getPrice());
    }

}
```

Rerun all the Maven tests target and our two tests should fail.

We go back to the ‘SimpleProductManager’ and implement the getter and setter methods for the products

property.

SimpleProductManager.java

```
1  package au.usyd.elec5619.service;
2
3  import java.util.List;
6
7  public class SimpleProductManager implements ProductManager {
8
9  private List<Product> products;
10
11     public List<Product> getProducts() {
12         return products;
13     }
14
15     public void increasePrice(int percentage) {
16         // TODO Auto-generated method stub
17     }
18
19     public void setProducts(List<Product> products) {
20         this.products = products;
21     }
22
23 }
```

Rerun the Maven tests target and all our tests should pass.

We proceed by implementing the following tests for the increasePrice() method:

- The list of products is null and the method executes gracefully.
- The list of products is empty and the method executes gracefully.
- Set a price increase of 10% and check the increase is reflected in the prices of all the products in the list.

In the file '**springapp/src/test/java/au.usyd.elec5619.service/SimpleProductManagerTest.java'**

SimpleProductManagerTest.java | SimpleProductManager.java

```java
package au.usyd.elec5619.service;

import java.util.ArrayList;

public class SimpleProductManagerTest extends TestCase {

    private SimpleProductManager productManager;

    private List<Product> products;

    private static int PRODUCT_COUNT = 2;

    private static Double CHAIR_PRICE = new Double(20.50);
    private static String CHAIR_DESCRIPTION = "Chair";

    private static String TABLE_DESCRIPTION = "Table";
    private static Double TABLE_PRICE = new Double(150.10);

    private static int POSITIVE_PRICE_INCREASE = 10;

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
        products = new ArrayList<Product>();

        // stub up a list of products
        Product product = new Product();
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

        product = new Product();
        product.setDescription("Table");
        product.setPrice(TABLE_PRICE);
        products.add(product);

        productManager.setProducts(products);
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        assertNull(productManager.getProducts());
    }

```

```
    public void testGetProducts() {
        List<Product> products = productManager.getProducts();
        assertNotNull(products);
        assertEquals(PRODUCT_COUNT, productManager.getProducts().size());

        Product product = products.get(0);
        assertEquals(CHAIR_DESCRIPTION, product.getDescription());
        assertEquals(CHAIR_PRICE, product.getPrice());

        product = products.get(1);
        assertEquals(TABLE_DESCRIPTION, product.getDescription());
        assertEquals(TABLE_PRICE, product.getPrice());
    }

    public void testIncreasePriceWithNullListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(NullPointerException ex) {
            fail("Products list is null.");
        }
    }

    public void testIncreasePriceWithEmptyListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.setProducts(new ArrayList<Product>());
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(Exception ex) {
            fail("Products list is empty.");
        }
    }

    public void testIncreasePriceWithPositivePercentage() {
        productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        double expectedChairPriceWithIncrease = 22.55;
        double expectedTablePriceWithIncrease = 165.11;

        List<Product> products = productManager.getProducts();
        Product product = products.get(0);
        assertEquals(expectedChairPriceWithIncrease, product.getPrice());

        product = products.get(1);
        assertEquals(expectedTablePriceWithIncrease, product.getPrice());
    }

}
```

We return to SimpleProductManager to implement increasePrice().

In the file '**springapp/src/main/java/au.usyd.elec5619.service/SimpleProductManager.java'**

SimpleProductManagerTest.java | SimpleProductManager.java

```
1  package au.usyd.elec5619.service;
2
3  import java.util.List;
6
7  public class SimpleProductManager implements ProductManager {
8
9      private List<Product> products;
10
11     public List<Product> getProducts() {
12         return products;
13     }
14
15     public void increasePrice(int percentage) {
16         if (products != null) {
17             for (Product product : products) {
18                 double newPrice = product.getPrice().doubleValue() *
19                                   (100 + percentage)/100;
20                 product.setPrice(newPrice);
21             }
22         }
23     }
24
25     public void setProducts(List<Product> products) {
26         this.products = products;
27     }
28
29 }
```

Rerun the Maven tests target and all our tests should pass. *HURRAH* JUnit has a saying: "keep the bar green to keep the code clean." For those of you running the tests in an IDE and are new to unit testing, we hope you're feeling imbued with a sense of greater sense of confidence and certainty that the code is truly working as specified in the business rules specification and as you intend. We certainly do.

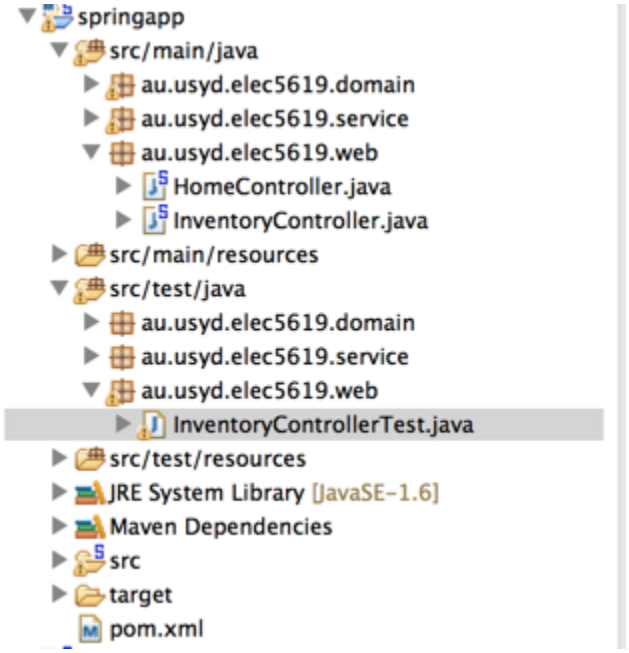
We're now ready to move back into the web layer to put a list of products into our Controller model.

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.



In the file '**springapp/src/main/java/au.usyd.elec5619.web/InventoryController.java**'

InventoryController.java

```java
package au.usyd.elec5619.web;

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

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

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import au.usyd.elec5619.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;
    }

}
```

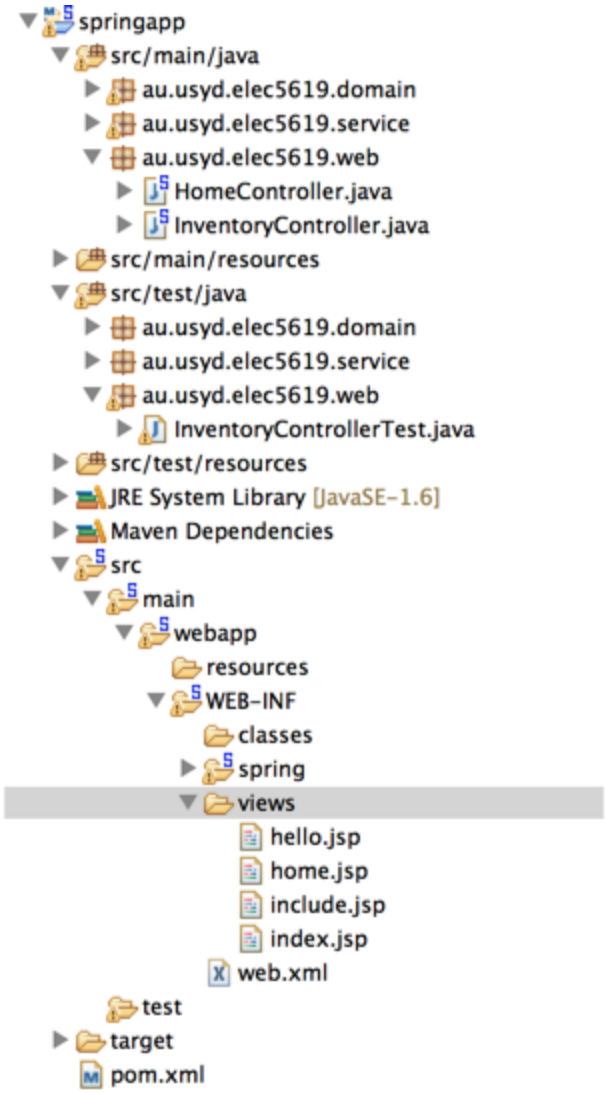
We will also need to create the InventoryControllerTest to supply a ProductManager and extract the value for 'now' from the model Map before the tests will pass again.

InventoryController.java | InventoryControllerTest.java

```
1  package au.usyd.elec5619.web;
2
3  import java.util.Map;
4
5  import junit.framework.TestCase;
6
7  import org.springframework.web.servlet.ModelAndView;
8
9  import au.usyd.elec5619.service.SimpleProductManager;
10
11 public class InventoryControllerTest extends TestCase {
12
13     public void testHandleRequestView() throws Exception{
14         InventoryController controller = new InventoryController();
15         controller.setProductManager(new SimpleProductManager());
16         ModelAndView modelAndView = controller.handleRequest(null, null);
17         assertEquals("hello", modelAndView.getViewName());
18         assertNotNull(modelAndView.getModel());
19         Map modelMap = (Map) modelAndView.getModel().get("model");
20         String nowValue = (String) modelMap.get("now");
21         assertNotNull(nowValue);
22     }
23 }
```

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.



In the file **'springapp/src/main/webapp/WEB-INF/views/hello.jsp'**

hello.jsp

```
1  <%@ include file="/WEB-INF/views/include.jsp" %>
2
3  <html>
4    <head><title><fmt:message key="title"/></title></head>
5    <body>
6      <h1><fmt:message key="heading"/></h1>
7      <p><fmt:message key="greeting"/> <c:out value="${model.now}"/></p>
8      <h3>Products</h3>
9      <c:forEach items="${model.products}" var="prod">
10       <c:out value="${prod.description}"/> <i>$<c:out value="${prod.price}"/></i><br><br>
11     </c:forEach>
12   </body>
13 </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 '**servlet-context.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.

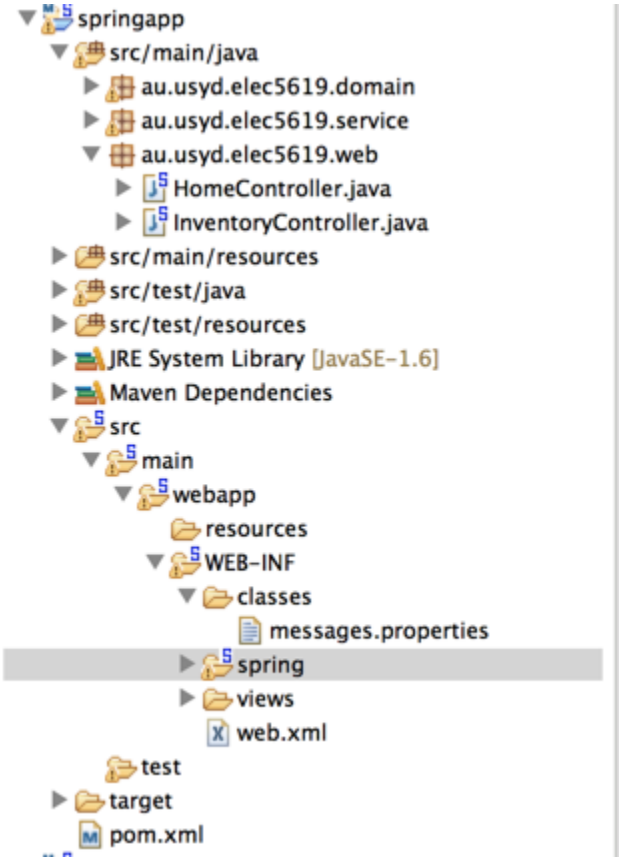
In the file '**springapp/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml**'

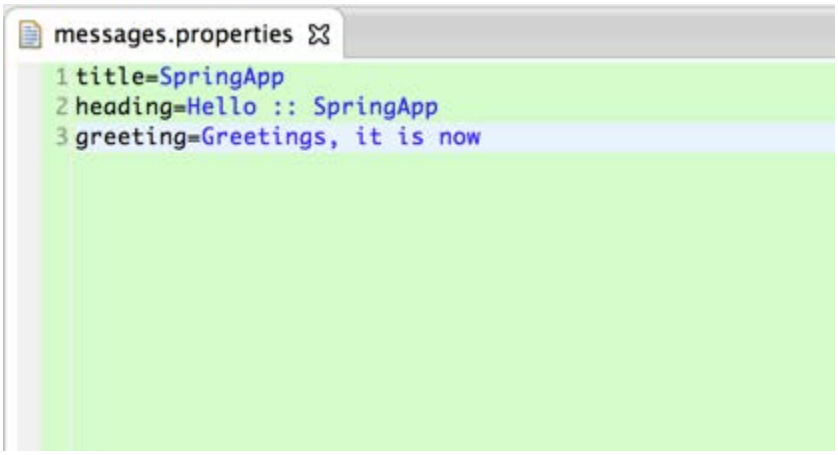
hello.jsp servlet-context.xml

```
<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources in the ${webappRo
<resources mapping="/resources/**" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
</beans:bean>

<context:component-scan base-package="au.usyd.elec5619" />

<beans:bean id="productManager" class="au.usyd.elec5619.service.SimpleProductManager">
    <beans:property name="products">
        <beans:list>
            <beans:ref bean="product1"/>
            <beans:ref bean="product2"/>
            <beans:ref bean="product3"/>
        </beans:list>
    </beans:property>
</beans:bean>

<!-- newly added stubs -->
<beans:bean id="product1" class="au.usyd.elec5619.domain.Product">
    <beans:property name="description" value="Lamp"/>
    <beans:property name="price" value="5.75"/>
</beans:bean>

<beans:bean id="product2" class="au.usyd.elec5619.domain.Product">
    <beans:property name="description" value="Table"/>
    <beans:property name="price" value="75.25"/>
</beans:bean>

<beans:bean id="product3" class="au.usyd.elec5619.domain.Product">
    <beans:property name="description" value="Chair"/>
    <beans:property name="price" value="22.79"/>
</beans:bean>

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

<beans:bean name="/hello.htm" class="au.usyd.elec5619.web.InventoryController">
    <beans:property name="productManager" ref="productManager"/>
</beans:bean>

</beans:beans>
```

4.4 Add the message bundle

We create a **'messages.properties'** file in the **'springapp/src/main/webapp/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'**.



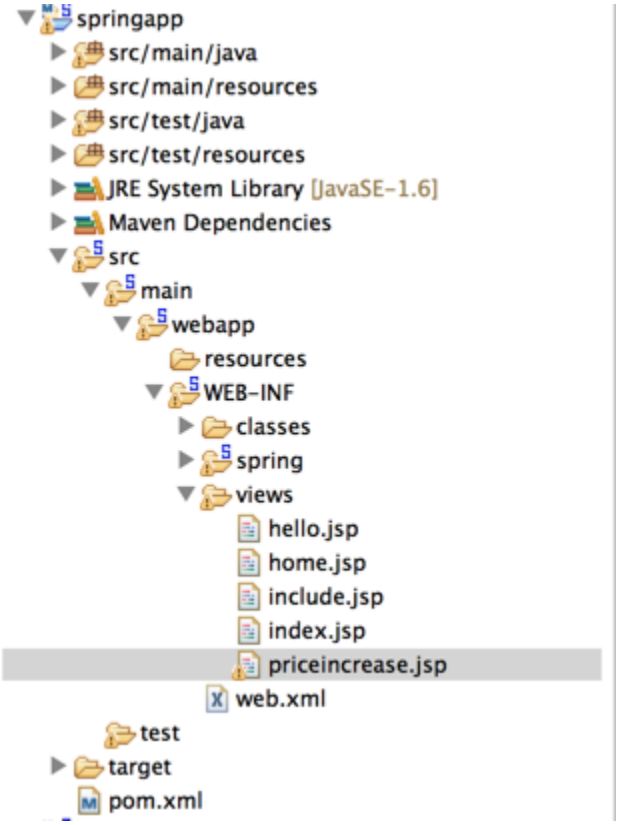


4.5 Adding a form

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.

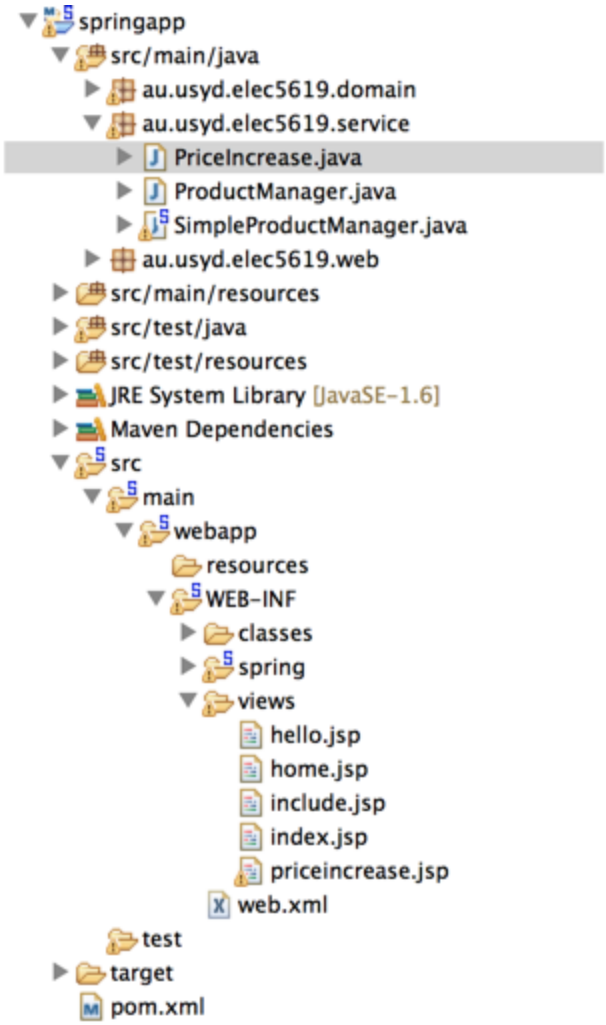
We also have to declare this taglib in a page directive in the jsp file, and then start using the tags we have thus imported. Add the JSP page **'priceincrease.jsp'** to the **'springapp/src/main/webapp/WEB-INF/views'** directory.



priceincrease.jsp

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

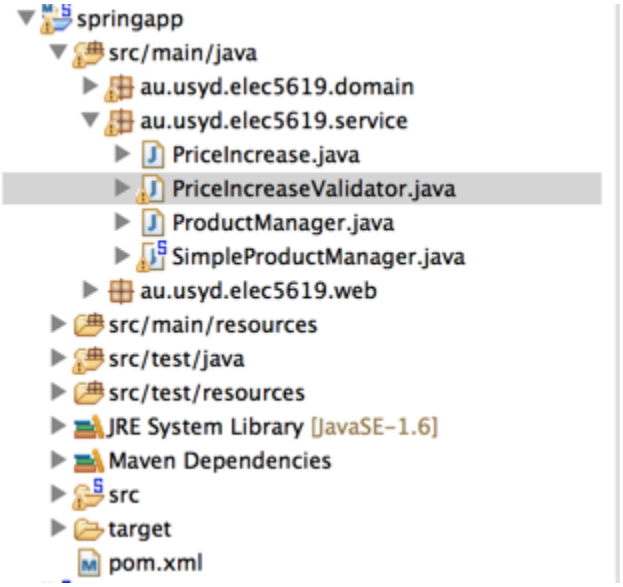
This next class is a very simple JavaBean class, and in our case there is a single property with a getter and setter. This is the object that the form will populate and that our business logic will extract the price increase percentage from.


springapp
src/main/java
au.usyd.elec5619.domain
au.usyd.elec5619.service
PriceIncrease.java
ProductManager.java
SimpleProductManager.java
au.usyd.elec5619.web
src/main/resources
src/test/java
src/test/resources
JRE System Library [JavaSE-1.6]
Maven Dependencies
src
main
webapp
resources
WEB-INF
classes
spring
views
hello.jsp
home.jsp
include.jsp
index.jsp
priceincrease.jsp
web.xml
test
target
pom.xml

priceincrease.jsp | PriceIncrease.java

```java
package au.usyd.elec5619.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.



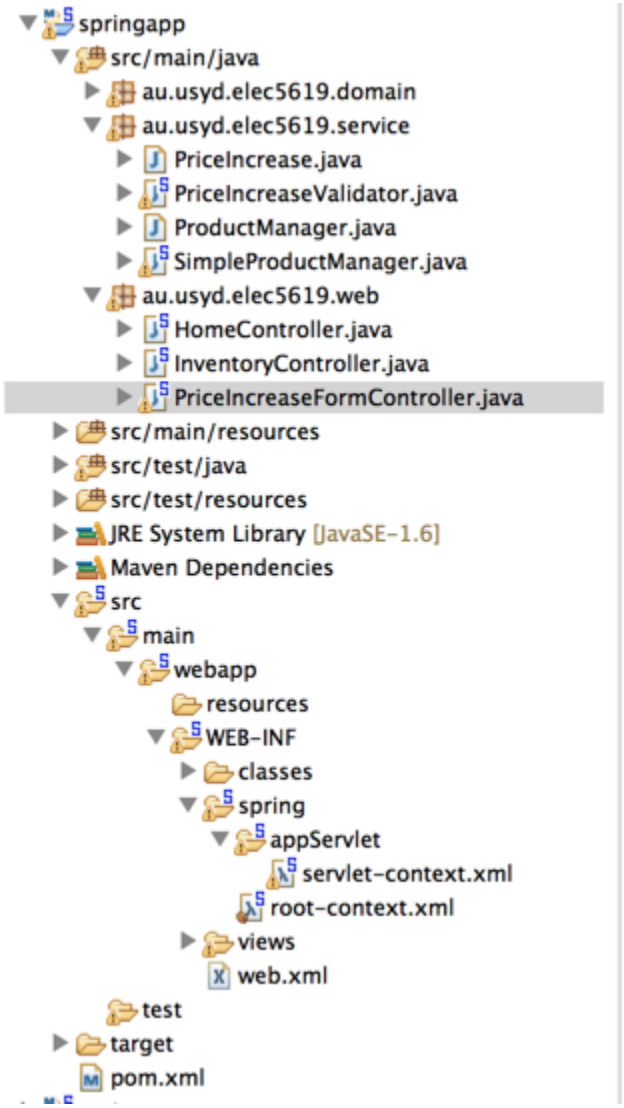
priceincrease.jsp | PriceIncrease.java | PriceIncreaseValidator.java

```
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 '**servlet-context.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.

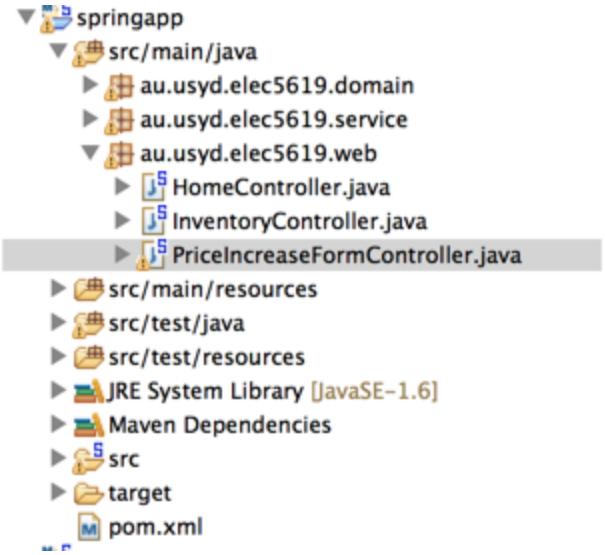


In the file '**springapp/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml**'

servlet-context.xml | PriceIncreaseFormController.java

```
            <beans:list>
                <beans:ref bean="product1"/>
                <beans:ref bean="product2"/>
                <beans:ref bean="product3"/>
            </beans:list>
        </beans:property>
    </beans:bean>

    <!-- 4.3 newly added stubs -->
    <beans:bean id="product1" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Lamp"/>
        <beans:property name="price" value="5.75"/>
    </beans:bean>

    <beans:bean id="product2" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Table"/>
        <beans:property name="price" value="75.25"/>
    </beans:bean>

    <beans:bean id="product3" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Chair"/>
        <beans:property name="price" value="22.79"/>
    </beans:bean>

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

    <beans:bean name="/hello.htm" class="au.usyd.elec5619.web.InventoryController">
        <beans:property name="productManager" ref="productManager"/>
    </beans:bean>

    <!-- 4.6 -->
    <beans:bean name="/priceincrease.htm" class="au.usyd.elec5619.web.PriceIncreaseFormController">
        <beans:property name="sessionForm" value="true"/>
        <beans:property name="commandName" value="priceIncrease"/>
        <beans:property name="commandClass" value="au.usyd.elec5619.service.PriceIncrease"/>
        <beans:property name="validator">
            <beans:bean class="au.usyd.elec5619.service.PriceIncreaseValidator"/>
        </beans:property>
        <beans:property name="formView" value="priceincrease"/>
        <beans:property name="successView" value="hello.htm"/>
        <beans:property name="productManager" ref="productManager"/>
    </beans:bean>

</beans: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/main/java
au.usyd.elec5619.domain
au.usyd.elec5619.service
au.usyd.elec5619.web
HomeController.java
InventoryController.java
PriceIncreaseFormController.java
src/main/resources
src/test/java
src/test/resources
JRE System Library [JavaSE-1.6]
Maven Dependencies
src
target
pom.xml

servlet-context.xml | PriceIncreaseFormController.java

```java
package au.usyd.elec5619.web;

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

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.view.RedirectView;

import au.usyd.elec5619.service.PriceIncrease;
import au.usyd.elec5619.service.ProductManager;

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.

servlet-context.xml | PriceIncreaseFormController.java | 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**'.

hello.jsp

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

    <!-- link to the increase price page -->
    <br>
        <a href="<c:url value="priceincrease.htm"/>">Increase Prices</a>
    <br>
  </body>
</html>
```

Chapter 5: Implementing Database Persistence

This is Part 5 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 and Part 4 developed the web interface. It is now time to introduce database persistence. We saw in the earlier parts how we loaded some business objects using bean definitions in a configuration file. It is obvious that this would never work in real life – whenever we re-start the server we are back to the original prices. We need to add code to actually persist these changes to a database.

5.1 Using database (MySQL) in a web application

Almost all web applications need to have their data persisted in databases, and for your project you will need to store your own data somewhere as well. This Chapter gives you a short introduction of using Hibernate and MySQL.

Normally, data manipulation within a database consists of create, retrieve, update, and delete (CRUD in short). Now we need to move our Products from the bean definition to a database.

Firstly, you need to install MySQL database in your computer. Here we provide several videos to guide you install it.

Install MySQL on Mac: https://www.youtube.com/watch?v=Tq0TXcH6dAU
Install MySQL on Windows: https://www.youtube.com/watch?v=O4xXzTIcnDE

This web page is a short tutorial to guide you how to do some operation in MySQL database: https://dev.mysql.com/doc/refman/5.7/en/database-use.html

After installing the MySQL database server. You will need to to create a database called “springapp” in you MySQL. Above link will help you to create databases in MySQL.

5.2 Dependency and configuration

Let’s start from resolving the dependencies of using Hibernate and MySQL. Because you will use new features in Java, you need to tell Maven what it need to get for you. Below is all required jar files (dependency) you’ll need for this chapter, and you can just copy and paste these dependencies to your pom.xml file.

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>3.1.0.RELEASE</version>
```

```
</dependency>

<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>3.5.6-Final</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>3.5.6-Final</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>3.5.6-Final</version>
</dependency>

<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
</dependency>
<dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
        <version>1.4</version>
</dependency>

<dependency>
       <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>4.3.1.Final</version>
 </dependency>
        <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>1.0.0.GA</version>
</dependency>
```

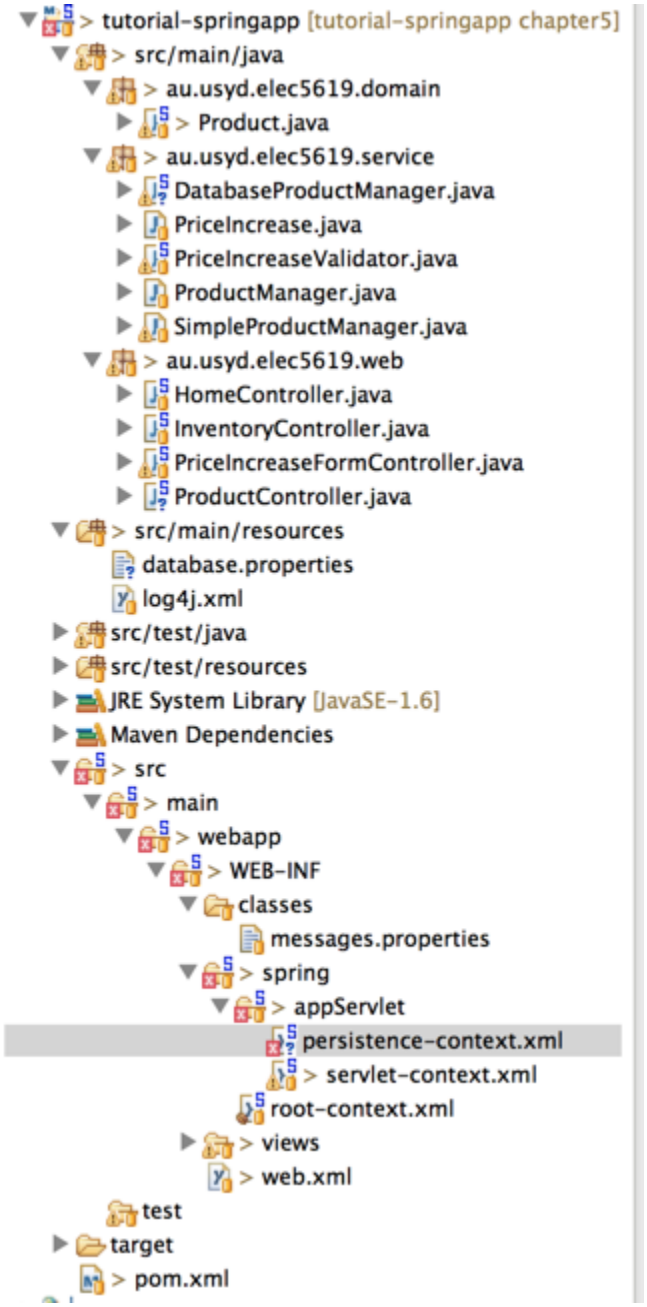
In order to use a database, you need to have all necessary database configurations values available. The below **database.properties** file should be placed in the **src/main/resources** folder

database.properties

```
jdbc.databaseName=springapp
jdbc.url=jdbc:mysql://localhost:3306/springapp?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
jdbc.username=root
jdbc.password=root
```

Make sure that the “jdbc.username” and “jdbc.password” are the same as yours when you installing your MySQL server.

Also, you need to tell Spring how to connect to a database. You need to have a **persistence-context.xml** file to do this for you.



persistence-context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
       ">

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:/database.properties"/>
    </bean>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url"
            value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan" value="au.usyd.elec5619" />
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</prop>
                <prop key="hibernate.show_sql">false</prop>
            </props>
        </property>
    </bean>

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <tx:annotation-driven proxy-target-class="true"/>
</beans>
```

And you need to tell your application that there is another xml configuration file to load while starting the application. You just need to add one line (line 25 of the below screenshot) in your web.xml file, which is in **src/main/webapp/WEB-INF** folder.

```
    <!-- Processes application requests -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                /WEB-INF/spring/appServlet/servlet-context.xml
                /WEB-INF/spring/appServlet/persistence-context.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

In Chapter 4.3, we created 3 products in **servlet-context.xml** (which is in **src/main/webapp/WEB-INF/spring/appServlet** folder) using bean initialization. Now, since we are going to use database, we can remove the 3 products from our application.

```

    <!-- 4.3 the SimpleProductManager -->
    <beans:bean id="productManager" class="au.usyd.elec5619.service.SimpleProductManager">
        <beans:property name="products">
            <beans:list>
                <beans:ref bean="product1"/>
                <beans:ref bean="product2"/>
                <beans:ref bean="product3"/>
            </beans:list>
        </beans:property>
    </beans:bean>

    <!-- 4.3 newly added stubs -->
    <beans:bean id="product1" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Lamp"/>
        <beans:property name="price" value="5.75"/>
    </beans:bean>

    <beans:bean id="product2" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Table"/>
        <beans:property name="price" value="75.25"/>
    </beans:bean>

    <beans:bean id="product3" class="au.usyd.elec5619.domain.Product">
        <beans:property name="description" value="Chair"/>
        <beans:property name="price" value="22.79"/>
    </beans:bean>

```

But the component that actually connects to the persistence layer is a bean with the name “productManager”, which is a type of SimpleProductManager in this case. For simplicity, we can just comment out the bean definition of this “productManager” and create a new one that connects to the HSQLDB with Hibernate.

```
25
26      <!-- 4.3 the SimpleProductManager -->
27      <!-- <beans:bean id="productManager" class="au.usyd.elec5619.service.SimpleProductManager">
28          <beans:property name="products">
29              <beans:list>
30                  <beans:ref bean="product1"/>
31                  <beans:ref bean="product2"/>
32                  <beans:ref bean="product3"/>
33              </beans:list>
34          </beans:property>
35      </beans:bean> -->
36
37      <!-- 4.3 newly added stubs -->
38      <beans:bean id="product1" class="au.usyd.elec5619.domain.Product">
39          <beans:property name="description" value="Lamp"/>
40          <beans:property name="price" value="5.75"/>
41      </beans:bean>
42
43      <beans:bean id="product2" class="au.usyd.elec5619.domain.Product">
44          <beans:property name="description" value="Table"/>
45          <beans:property name="price" value="75.25"/>
46      </beans:bean>
47
48      <beans:bean id="product3" class="au.usyd.elec5619.domain.Product">
49          <beans:property name="description" value="Chair"/>
50          <beans:property name="price" value="22.79"/>
51      </beans:bean>
```

5.3 Annotation based web application development (**Important**)

So far (chapter 2 to 4), all the tutorial is done using xml configuration, which is verbose. A feature of Java called Annotation is able to accomplish the same task. For web development, typically there are 3 annotations that are used most often, which are @Entity, @Controller, and @Service. (Using XML, all you have are beans, but with Annotations, the "beans" are "categorized" into different classes based on their functionality)

The name of these annotations are derived from the roles of the corresponding components within the MVC design pattern.

Entity

@Entity announces the annotated class to be a domain entity, which is the M (model) in MVC. In our case, the Product class should be annotated with @Entity. As you can see in the below screenshot, there are other annotations within the Product class. Those annotations are used by Hibernate to map this Product class to a table called "Product" in the database.

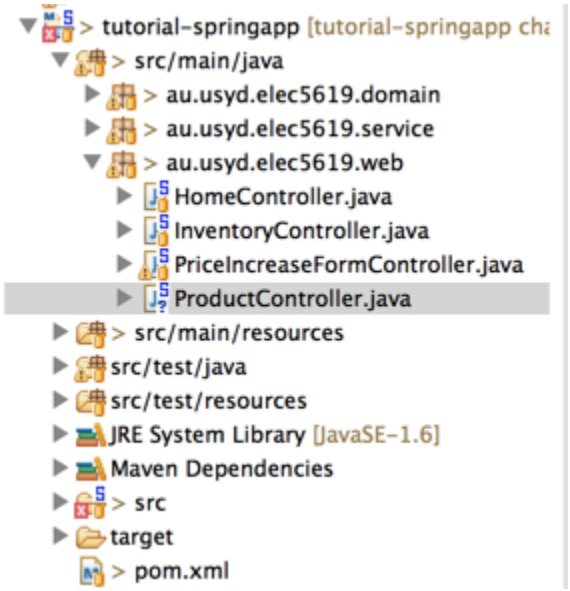
Your task is to annotate Product class (which you have already got in the domain package) in the same way as the screenshot.

Product.java

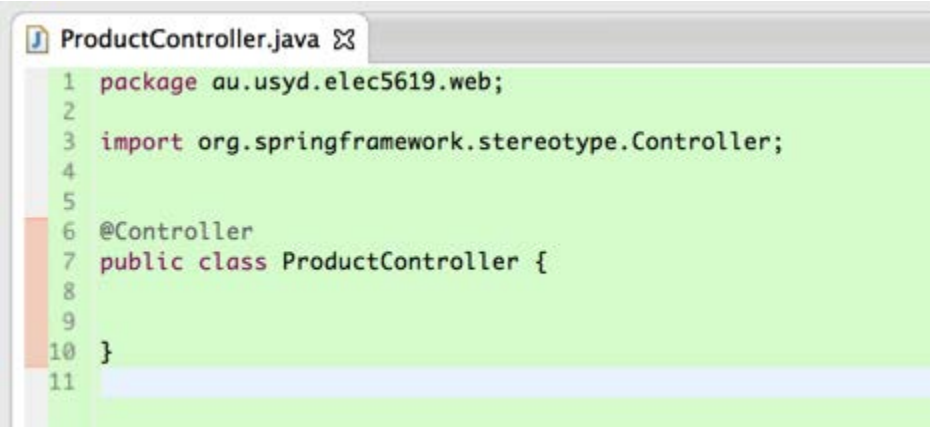
```java
package au.usyd.elec5619.domain;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="Product")
public class Product implements Serializable {

    @Id
    @GeneratedValue
    @Column(name="Id")
    private long id;

    @Column(name="Description")
    private String description;

    @Column(name="Price")
    private Double price;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Description: " + description + ";");
        buffer.append("Price: " + price);
        return buffer.toString();
    }
}
```

Controller

@Controller is used to define a controller, which is C in MVC. We need to create a new controller called **ProductController** to handle the CRUD functions of products.



In the **au.usyd.elec5619.web** package, create a new Java class with the name "ProductController", and at the class level, annotate it with @Controller.



Now, what you have is a controller, and if you run your application (actually, it won't run, because we have commented out the productManager bean. If you uncomment it, this application will run in the same way as before), Spring will know that this ProductController is a controller, and when requests come, the DispatcherServlet should dispatch some requests to this controller based on the given request URL. However, you haven't given any URL mapping configuration information to this controller.

ProductController.java

```
1  package au.usyd.elec5619.web;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5
6
7  @Controller
8  public class ProductController {
9
10     public String addProduct(Model uiModel) {
11
12         return "add";
13     }
14 }
15
```

Now let's add a method to this controller called "addProduct", which should handle the add-a-product request send from a browser. But so far, Spring still doesn't know how to map the request URL to this controller.

Now we add a new annotation @RequestMapping at both the class level and the method level. And based the annotation at line 9 and line 12, Spring will be able to map a request to http://localhost:8080/elec5619/product/add to ProductController.addProduct(). And this method simply return the logical name of the view file.

ProductController.java

```
1  package au.usyd.elec5619.web;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6
7
8  @Controller
9  @RequestMapping(value="/product/**")
10 public class ProductController {
11
12     @RequestMapping(value="/add")
13     public String addProduct(Model uiModel) {
14
15         return "add";
16     }
17 }
18
```

Now you need to create a new view file for the addProduct function called **add.jsp**.

ProductController.java | add.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Add a new product on this page</title>
    </head>
    <body>
        <h1>Hi, you can add a new product on this page</h1>
        <form action="add" method="post">
            Description: <input type="text" name="description"/>
            Price: <input type="text" name="price"/>
            <input type="submit" value="Add"/>
        </form>
    </body>
</html>
```

This view file simply contains a form for creating a new Product. And this form will be submitted to http://localhost:8080/elec5619/product/add using POST method, instead of GET.

The addProduct() method we have now is used to handle GET request, which means we need to create another method to handle POST request to the same URL.

ProductController.java | add.jsp

```
package au.usyd.elec5619.web;

import javax.servlet.http.HttpServletRequest;

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


@Controller
@RequestMapping(value="/product/**")
public class ProductController {

    @RequestMapping(value="/add")
    public String addProduct(Model uiModel) {

        return "add";
    }

    @RequestMapping(value="/add", method=RequestMethod.POST)
    public String addProduct(HttpServletRequest httpServletRequest) {

        return "redirect:/hello.htm";
    }
}

```

See the difference with the previous method? The new addProduct() method is specifically configured to handle POST request through the **method=RequestMethod.POST** setting. And this method simply redirect the user to **hello.htm** which contains all existing products in the database.

Service

@Service is used to define a service for your web application. A service is used to handle all processing of any model (or business logic), and a service is typically used inside a controller.

We need to create a new ProductManager subclass for the Product to handle the low level (closer to database) CRUD.

Inside the **au.usyd.elec5619.service** package, create a new class called **DatabaseProductManager** and it should implement the **ProductManager** interface. Wait a minute, there will be new method in the **DatabaseProductManager,** which handle the CRUD of a Product, do they go into the **ProductManager** interface? Yes.

We need to update the **ProductManager** interface first.

ProductManager.java

```
1  package au.usyd.elec5619.service;
2
3  import java.io.Serializable;
4  import java.util.List;
5
6  import au.usyd.elec5619.domain.Product;
7
8  public interface ProductManager extends Serializable{
9
10     public void increasePrice(int percentage);
11
12     public List<Product> getProducts();
13
14     public void addProduct(Product product);
15
16     public Product getProductById(long id);
17
18     public void updateProduct(Product product);
19
20     public void deleteProduct(long id);
21
22 }
```

Then, we can create the new **DatabaseProductManager** class to implement the updated interface.

DatabaseProductManager.java

```java
package au.usyd.elec5619.service;

import java.util.List;

import au.usyd.elec5619.domain.Product;


public class DatabaseProductManager implements ProductManager {

    @Override
    public void increasePrice(int percentage) {
        // TODO Auto-generated method stub

    }

    @Override
    public List<Product> getProducts() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void addProduct(Product product) {
        // TODO Auto-generated method stub

    }

    @Override
    public Product getProductById(long id) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void updateProduct(Product product) {
        // TODO Auto-generated method stub

    }

    @Override
    public void deleteProduct(long id) {
        // TODO Auto-generated method stub

    }
}

```

There will be errors in the **SimpleProductManager** class, because we haven't implemented the new methods added to the **ProductManager** interface. You can just add them to the **ProductManager** class and leave them empty.

SimpleProductManager.java

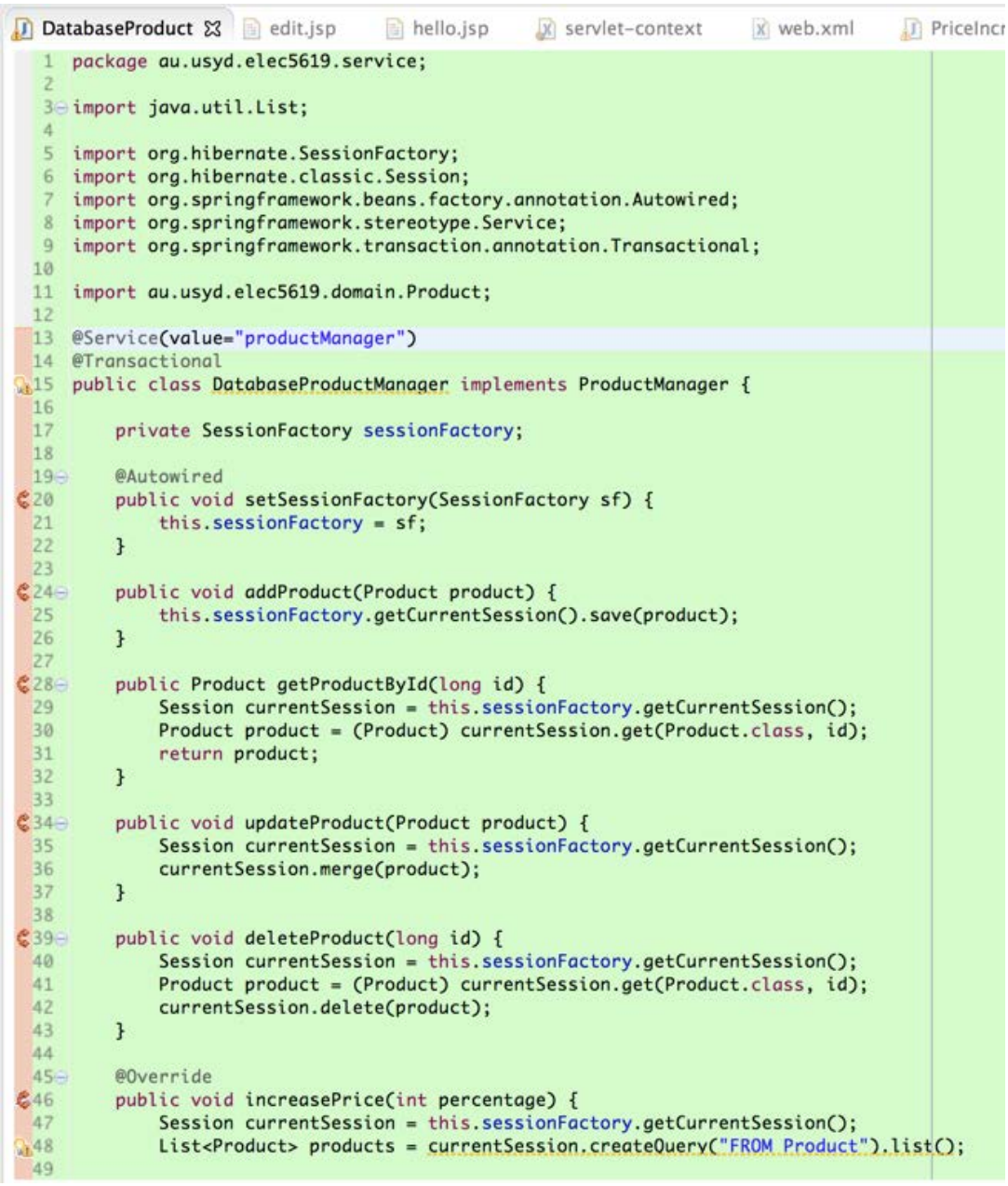
```
import java.util.List;

public class SimpleProductManager implements ProductManager {

    private List<Product> products;

    public List<Product> getProducts() {
        return products;
    }

    public void increasePrice(int percentage) {
        if (products != null) {
            for (Product product : products) {
                double newPrice = product.getPrice().doubleValue() *
                                  (100 + percentage)/100;
                product.setPrice(newPrice);
            }
        }
    }

    public void setProducts(List<Product> products) {
        this.products = products;
    }

    @Override
    public void addProduct(Product product) {
        // TODO Auto-generated method stub

    }

    @Override
    public Product getProductById(long id) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void updateProduct(Product product) {
        // TODO Auto-generated method stub

    }

    @Override
    public void deleteProduct(long id) {
        // TODO Auto-generated method stub

    }

}
```

Since it is a service class, we annotate it with the @Service annotation, and now Spring knows that this **DatabaseProductManager** is a Service.

Now, we should introduce Hibernate to our code. Hibernate is like the superior of JDBC at a higher level, and in Hibernate, the Session is the most important entity. The session is all you have to access the database.

To get access to the "session" you need a SessionFactory, which you have already defined at the bottom of the **persistence-context.xml** file. The **SessionFactory** is a private instance variable of **DatabaseProductManager** and there is a **setSessionFactory()** function with the annotation **@Autowired**, which is able to inject an instance of **SessionFactory** to this **DatabaseProductManager** when this **DatabaseProductManager** is being created. This process is called "dependency injection".

The **@Transactional** annotation at the class level is used to handle database transactions.



```
        if (products != null) {
            for (Product product : products) {
                double newPrice = product.getPrice().doubleValue() *
                                  (100 + percentage)/100;
                product.setPrice(newPrice);
                currentSession.save(product);
            }
        }
    }

    @Override
    public List<Product> getProducts() {
        return this.sessionFactory.getCurrentSession().createQuery("FROM Product").list();
    }

}
```

I've mentioned that a service is typically used in a controller (or other services), and now let's update our **ProductController**.

ProductController.java

```
package au.usyd.elec5619.web;

import javax.annotation.Resource;


@Controller
@RequestMapping(value="/product/**")
public class ProductController {

    @Resource(name="productManager")
    private ProductManager productManager;

    @RequestMapping(value="/add")
    public String addProduct(Model uiModel) {

        return "add";
    }

    @RequestMapping(value="/add", method=RequestMethod.POST)
    public String addProduct(HttpServletRequest httpServletRequest) {

        Product product = new Product();
        product.setDescription(httpServletRequest.getParameter("description"));
        product.setPrice(Double.valueOf(httpServletRequest.getParameter("price")));
        this.productManager.addProduct(product);

        return "redirect:/hello.htm";
    }

```

What I did was inject an instance of the **ProductManager** into the controller and use this productManager to persist a new product in the database. (In this case, the **ProductManager** is actually an instance of **DatabaseProductManager**)

Now you can run your application and add new products.

5.3 Others

All CRUD requests should be handled in the **ProductController**, so below is the complete code.

ProductController.java

```java
package au.usyd.elec5619.web;

import javax.annotation.Resource;
import javax.servlet.http.HttpServletRequest;
import javax.validation.Valid;

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

import au.usyd.elec5619.domain.Product;
import au.usyd.elec5619.service.ProductManager;


@Controller
@RequestMapping(value="/product/**")
public class ProductController {

    @Resource(name="productManager")
    private ProductManager productManager;

    @RequestMapping(value="/add")
    public String addProduct(Model uiModel) {

        return "add";
    }

    @RequestMapping(value="/add", method=RequestMethod.POST)
    public String addProduct(HttpServletRequest httpServletRequest) {

        Product product = new Product();
        product.setDescription(httpServletRequest.getParameter("description"));
        product.setPrice(Double.valueOf(httpServletRequest.getParameter("price")));
        this.productManager.addProduct(product);

        return "redirect:/hello.htm";
    }

    @RequestMapping(value="/edit/{id}", method=RequestMethod.GET)
    public String editProduct(@PathVariable("id") Long id, Model uiModel) {

        Product product = this.productManager.getProductById(id);
        uiModel.addAttribute("product", product);

        return "edit";
    }

    @RequestMapping(value="/edit/**", method=RequestMethod.POST)
    public String editProduct(@Valid Product product) {

        this.productManager.updateProduct(product);
        System.out.println(product.getId());

        return "redirect:/hello.htm";
    }

    @RequestMapping(value="/delete/{id}", method=RequestMethod.GET)
    public String deleteProduct(@PathVariable("id") Long id) {

        this.productManager.deleteProduct(id);

        return "redirect:/hello.htm";
    }
}
```

For Product editing, you need to create a edit.jsp page.

ProductController.java | add.jsp | DatabaseProductManager.java | edit.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Edit an old product on this page</title>
    </head>
    <body>
        <h1>You can edit the product shown below</h1>
        <sf:form method="POST" modelAttribute="product">
            <fieldset>
                <table>
                    <tr>
                        <th><label for="product_description">Description:</label></th>
                        <td><sf:input path="description"/></td>
                    </tr>
                    <tr>
                        <th><label for="product_price">Price</label></th>
                        <td><sf:input path="price"/></td>
                    </tr>
                    <tr>
                        <th><a href="hello.htm"><button>Cancel</button></a></th>
                        <!-- This hidden field is required for Hibernate to recognize this Product -->
                        <td><sf:hidden path="id"/>
                        <td><input type="submit" value="Done"/></td>
                    </tr>
                </table>
            </fieldset>
        </sf:form>
    </body>
</html>
```

The hello.jsp should also be updated to provide links to edit and delete products.

ProductController.java | add.jsp | DatabaseProductManager.java | edit.jsp | hello.jsp

```
<%@ include file="/WEB-INF/views/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="${prod.price}" /></i>
            <a href="product/edit/${prod.id }">edit</a>
            <a href="product/delete/${prod.id }">delete</a>
            <br>
            <br>
        </c:forEach>

        <!-- link to the increase price page -->
        <br>
        <a href="<c:url value="priceincrease.htm"/>">Increase Prices</a>
        <br>
    </body>
</html>
```