Spring

Introduction to Spring MVC

Developing a Spring Framework MVC application step-by-step

Version 2.5

Table of Contents

Overview

This document is a step-by-step guide on how to develop a web application from scratch using the Spring Framework.

Only a cursory knowledge of Spring itself is assumed, and as such this tutorial is ideal if you are learning or investigating Spring. Hopefully by the time you have worked your way through the tutorial material you will see how the constituent parts of the Spring Framework, namely Inversion of Control (IoC), Aspect-Oriented Programming (AOP), and the various Spring service libraries (such as the JDBC library) all fit together in the context of a Spring MVC web application.

Spring provides several options for configuring your application. The most popular one is using XML files. This is also the traditional way that has been supported from the first release of Spring. With the introduction of Annotations in Java 5, we now have an alternate way of configuring our Spring applications. The new Spring 2.5 release introduces extensive support for using Annotations to configure a web application.

This document uses the traditional XML style for configuration. We are working on an *"Annotation Edition"* of this document and hope to publish it in the near future.

Please note that we are not going to cover any background information or theory in this tutorial; there are plenty of books available that cover the areas in depth; whenever a new class or feature is used in the tutorial, forward pointers to the relevant section(s) of the Spring reference documentation are provided where the class or feature is covered in depth.

1. What's covered

The following list details all of the various parts of the Spring Framework that are covered over the course of the tutorial.

- Inversion of Control (IoC)

- The Spring Web MVC framework

- Data access with JDBC

- Unit and integration testing

- Transaction management

2. Prerequisite software

The following prerequisite software and environment setup is assumed. You should also be reasonably comfortable using the following technologies.

- Java SDK 1.5

- Ant 1.7

- Apache Tomcat 6.0.14

- Eclipse 3.3 (Recommended, but not necessary)

Eclipse 3.3 Europa (http://www.eclipse.org/europa) with the Web Tools Platform (WTP) Project (http://www.eclipse.org/webtools) and the Spring IDE Project (http://www.springide.org) provides an excellent envirnment for web development.

You may of course use pretty much any variation or version of the above software. If you want to use NetBeans or IntelliJ instead of Eclipse or Jetty instead of Tomcat, then many of the tutorial steps will not translate directly to your environment but you should be able to follow along anyway.

3. The application we are building

The application we will be building from scratch over the course of this tutorial is a *very basic* inventory management system. This inventory management system is severely constrained in terms of scope; find below a use case diagram illustrating the simple use cases that we will be implementing. The reason why the application is so constrained is so that you can concentrate on the specifics of Spring Web MVC and Spring, and not the finer details of inventory management.
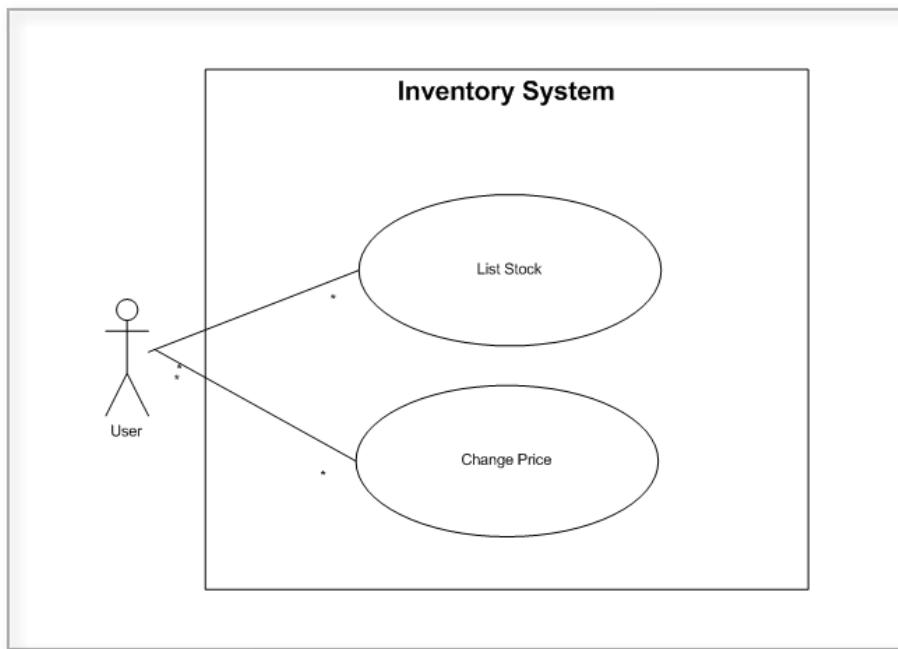
Use case diagram of the inventory management system

We will start by setting up the basic project directory structure for our application, downloading the required libraries, setting up our Ant build scripts, etc. The first step gives us a solid foundation on which to develop the application proper in parts 2, 3, and 4.

Once the basic setup is out of the way, Spring itself will be introduced, starting with the Spring Web MVC framework. We will use Spring Web MVC to display the inventoried stock, which will involve writing some simple Java classes and some JSPs. We will then move onto introducing persistent data access into our application, using Spring's Simple JDBC support.

By the time we have finished all of the steps in the tutorial, we will have an application that does basic inventory management, including listing stock and permitting the price increase of such stock.

Chapter 1. Basic Application and Environment Setup

1.1. Create the project directory structure

We are going to need a place to keep all the source and other files we will be creating, so let's create a directory named `'springapp'`. The decision as to where you create this directory is totally up to you; we created ours in a `'Projects'` directory that we already had in our `'home'` directory so the complete path to our project directory is now `'$HOME/Projects/springapp'`. Inside this directory we create a sub-directory named `'src'` to hold all the Java source files that we are going to create. Then we create another sub-directory that we name `'war'`. This directory will hold everything that should go into the WAR file that we will use to package and deploy our application. All source files other than Java source, like JSPs and configuration files, belong in the `'war'` directory.

Find below a screen shot of what your project directory structure must look like after following the above instructions. *(The screen shot shows the project directory structure inside the Eclipse IDE: you do not need to use the Eclipse IDE to complete this tutorial successfully, but using Eclipse will make it much easier to follow along.)*
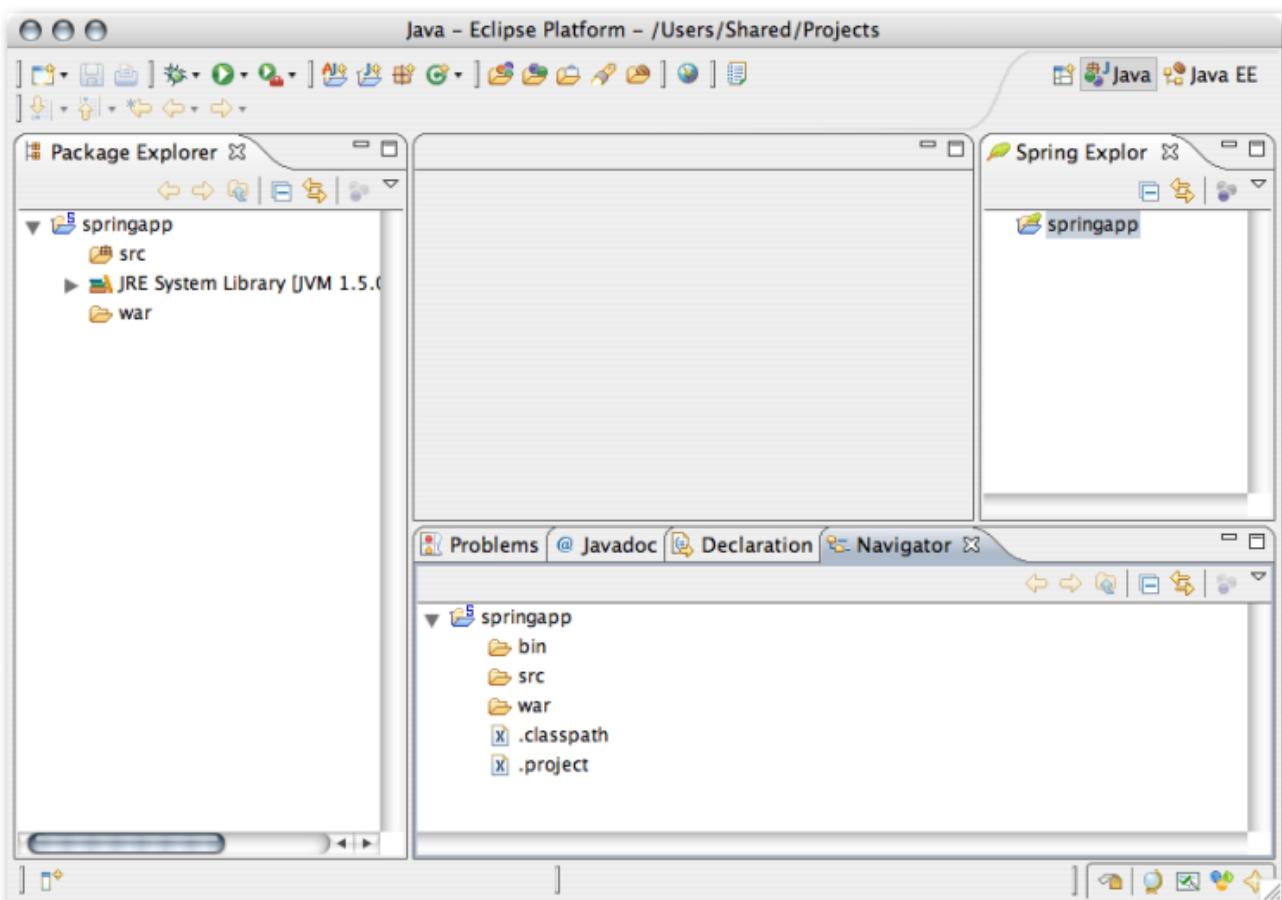
The project directory structure

1.2. Create `'index.jsp'`

Since we are creating a web application, let's start by creating a very simple JSP page named `'index.jsp'` in

the `'war'` directory. The `'index.jsp'` is the entry point for our application.

`'springapp/war/index.jsp'`:

```
<html>
  <head><title>Example :: Spring Application</title></head>
  <body>
    <h1>Example - Spring Application</h1>
    <p>This is my test.</p>
  </body>
</html>
```

Just to have a complete web application, let's create a `'WEB-INF'` directory inside the `'war'` directory and place a `'web.xml'` file in this new directory.

`'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" >

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

</web-app>
```

1.3. Deploy the application to Tomcat

Let's now write the Ant build script that we are going to use throughout the tutorial. This Ant build script will contain targets for compiling, building and deploying the application. A separate build script will be used for application server specific targets, such as targets for controlling the application under Tomcat.

`'springapp/build.xml'`:

```
<?xml version="1.0"?>

<project name="springapp" basedir="." default="usage">
    <property file="build.properties"/>

    <property name="src.dir" value="src"/>
    <property name="web.dir" value="war"/>
    <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
    <property name="name" value="springapp"/>

    <path id="master-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <!-- We need the servlet API classes: -->
        <!--  * for Tomcat 5/6 use servlet-api.jar -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="servlet*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
    </path>

    <target name="usage">
        <echo message=""/>
        <echo message="${name} build file"/>
        <echo message="----------------------------------"/>
```

```
        <echo message=""/>
        <echo message="Available targets are:"/>
        <echo message=""/>
        <echo message="build      --> Build the application"/>
        <echo message="deploy     --> Deploy application as directory"/>
        <echo message="deploywar --> Deploy application as a WAR file"/>
        <echo message="install    --> Install application in Tomcat"/>
        <echo message="reload     --> Reload application in Tomcat"/>
        <echo message="start      --> Start Tomcat application"/>
        <echo message="stop       --> Stop Tomcat application"/>
        <echo message="list       --> List Tomcat applications"/>
        <echo message=""/>
    </target>

    <target name="build" description="Compile main source tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
               deprecation="false" optimize="false" failonerror="true">
            <src path="${src.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <target name="deploy" depends="build" description="Deploy application">
        <copy todir="${deploy.path}/${name}" preservelastmodified="true">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </copy>
    </target>

    <target name="deploywar" depends="build" description="Deploy application as a WAR file">
        <war destfile="${name}.war"
             webxml="${web.dir}/WEB-INF/web.xml">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </war>
        <copy todir="${deploy.path}" preservelastmodified="true">
            <fileset dir=".">
                <include name="*.war"/>
            </fileset>
        </copy>
    </target>

<!-- ============================================================ -->
<!-- Tomcat tasks - remove these if you don't have Tomcat installed -->
<!-- ============================================================ -->

    <path id="catalina-ant-classpath">
        <!-- We need the Catalina jars for Tomcat -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="catalina-ant.jar"/>
        </fileset>
    </path>

    <taskdef name="install" classname="org.apache.catalina.ant.InstallTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="start" classname="org.apache.catalina.ant.StartTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="stop" classname="org.apache.catalina.ant.StopTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>

    <target name="install" description="Install application in Tomcat">
        <install url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"
                 path="/${name}"
```

```
                            war="${name}"/>
    </target>

    <target name="reload" description="Reload application in Tomcat">
        <reload url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
    </target>

    <target name="start" description="Start Tomcat application">
        <start url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
    </target>

    <target name="stop" description="Stop Tomcat application">
        <stop url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
    </target>

    <target name="list" description="List Tomcat applications">
        <list url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"/>
    </target>

<!-- End Tomcat tasks -->

</project>
```

If you are using a different web application server, then you can remove the Tomcat specific tasks at the end of the build script. You will have to rely on your server's hot deploy feature, or you will have to stop and start your application manually.

If you are using an IDE, you may find a number of errors reported by the IDE in the `build.xml` such as the Tomcat targets. You can ignore these. The file listing above is correct.

The above Ant build script now contains all the targets that we are going to need to make our development efforts easier. We are not going to cover this script in detail, since most if not all of it is pretty much standard Ant and Tomcat stuff. You can just copy the above build file text and paste it into a new file called `build.xml` in the root of your development directory tree. We also need a `build.properties` file that you should customize to match your server installation. This file belongs in the same directory as the `build.xml` file.

`'springapp/build.properties'`:

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret
```

*If you are on a system where you are not the owner of the Tomcat installation, then the Tomcat owner must either grant you full access to the `'webapps'` directory or the owner must create a new directory named `'springapp'` in the `'webapps'` directory of the Tomcat installation directory, and also give you full rights to deploy to this newly created directory. On Linux, run the command **'chmod a+rwx springapp'** to give*

everybody full rights to this directory.

To create Tomcat user named 'tomcat' with 's3cret' as their password, go to the Tomcat users file `'appserver.home/conf/tomcat-users.xml'` *and add the user entry.*

```xml
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="s3cret" roles="manager"/>
</tomcat-users>
```

Now we run Ant to make sure that everything is working okay. You must have your current directory set to the `'springapp'` directory.

Open up a command shell (or prompt) and execute **'ant'** .

```
$ ant
Buildfile: build.xml

usage:
     [echo]
     [echo] springapp build file
     [echo] ----------------------------------
     [echo]
     [echo] Available targets are:
     [echo]
     [echo] build     --> Build the application
     [echo] deploy    --> Deploy application as directory
     [echo] deploywar --> Deploy application as a WAR file
     [echo] install   --> Install application in Tomcat
     [echo] reload    --> Reload application in Tomcat
     [echo] start     --> Start Tomcat application
     [echo] stop      --> Stop Tomcat application
     [echo] list      --> List Tomcat applications
     [echo]

BUILD SUCCESSFUL
Total time: 2 seconds
```

The last thing we need to do here is to build and deploy the application. Just run Ant and specify `'deploy'` or `'deploywar'` as the target.

```
$ ant deploy
Buildfile: build.xml

build:
    [mkdir] Created dir: /Users/trisberg/Projects/springapp/war/WEB-INF/classes

deploy:
     [copy] Copying 2 files to /Users/trisberg/apache-tomcat-5.5.17/webapps/springapp

BUILD SUCCESSFUL
Total time: 4 seconds
```

1.4. Check the application works

Let's just quickly start Tomcat by running `'${appserver.home}/bin/startup.bat'`. To make sure that we can access the application, run the `'list'` task from our build file to see if Tomcat has picked up the new application.

```
$ ant list
Buildfile: build.xml

list:
```

```
    [list] OK - Listed applications for virtual host localhost
    [list] /springapp:running:0:springapp
    [list] /manager:running:0:manager
    [list] /:running:0:ROOT
    [list] /docs:running:0:docs
    [list] /examples:running:0:examples
    [list] /host-manager:running:0:host-manager

BUILD SUCCESSFUL
Total time: 3 seconds
```

You can now open up a browser and navigate to the starting page of our application at the following URL: http://localhost:8080/springapp/index.jsp.
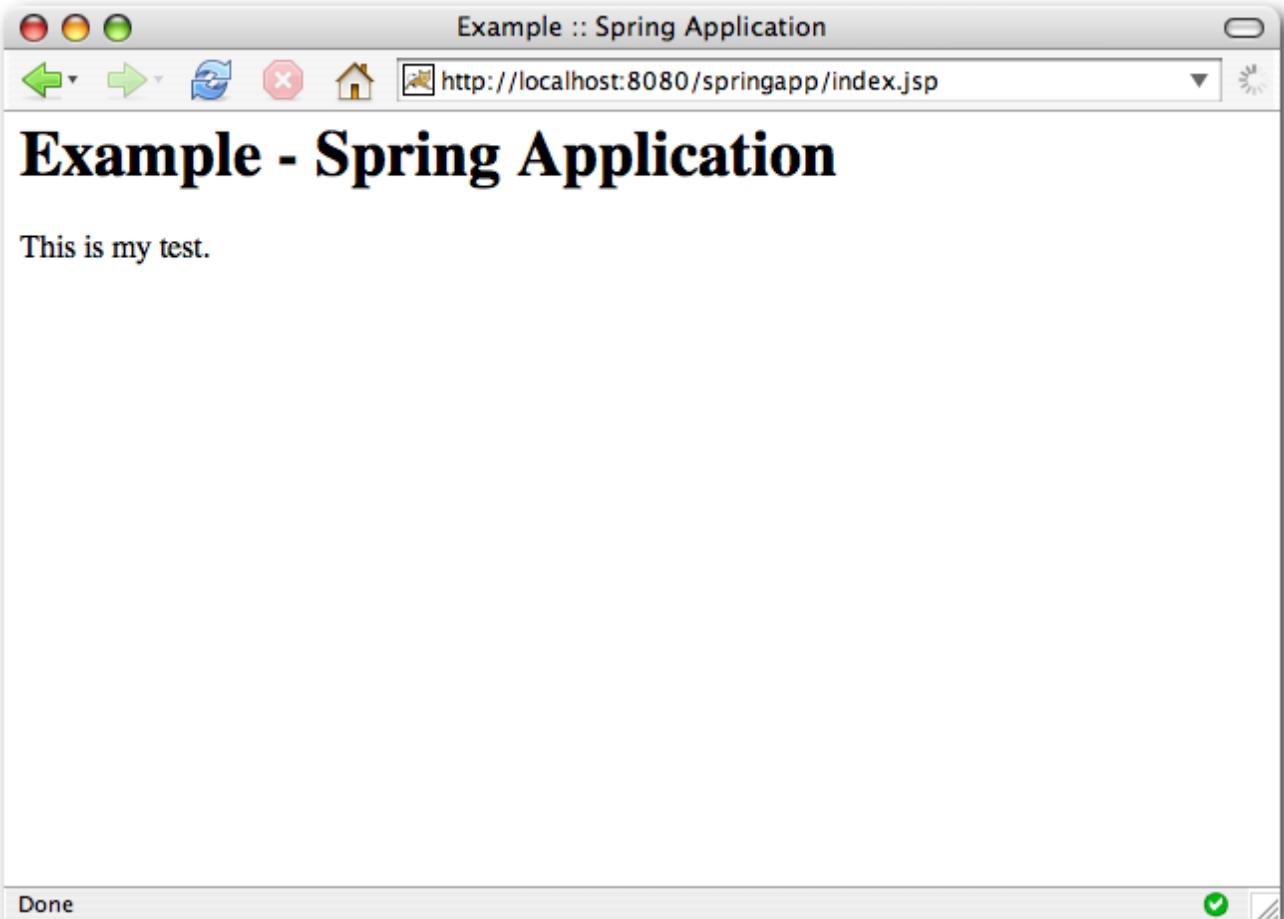
The application's starting page

1.5. Download the Spring Framework

If you have not already downloaded the Spring Framework, now is the time to do so. We are currently using the 'Spring Framework 2.5' release that can be downloaded from http://www.springframework.org/download. Unzip this file somewhere as we are going to use several files from this download later on.

This completes the setup of the environment that is necessary, and now we can start actually developing our Spring Framework MVC application.

1.6. Modify `web.xml` in the `WEB-INF` directory

Go to the `'springapp/war/WEB-INF'` directory. Modify the minimal `'web.xml'` file that we created earlier. We will define a `DispatcherServlet` (also known as a `'Front Controller'` (Crupi et al)). It is going to control where all our requests are routed based on information we will enter at a later point. This servlet definition also has an attendant `<servlet-mapping/>` entry that maps to the URL patterns that we will be using. We have decided to let any URL with an `'.htm'` extension be routed to the `'springapp'` servlet (the `DispatcherServlet`).

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

```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>

</web-app>
```

Next, create a file called `'springapp-servlet.xml'` in the `'springapp/war/WEB-INF'` directory. This file contains the bean definitions (plain old Java objects) used by the `DispatcherServlet`. It is the `WebApplicationContext` where all web-related components go. The name of this file is determined by the value of the `<servlet-name/>` element from the `'web.xml'`, with `'-servlet'` appended to it (hence `'springapp-servlet.xml'`). This is the standard naming convention used with Spring's Web MVC framework. Now, add a bean entry named `'/hello.htm'` and specify the class as `springapp.web.HelloController`. This defines the controller that our application will be using to service a request with the corresponding URL mapping of `'/hello.htm'`. The Spring Web MVC framework uses an implementation class of the interface called `HandlerMapping` to define the mapping between a request URL and the object that is going to handle that request (the handler). Unlike the `DispatcherServlet`, the `HelloController` is responsible for handling a request for a particular page of the website and is also known as a `'Page Controller'` (Fowler). The default `HandlerMapping` that the `DispatcherServlet` uses is the `BeanNameUrlHandlerMapping`; this class will use the bean name to map to the URL in the request so that the `DispatcherServlet` knows which controller must be invoked for handling different URLs.

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

```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 name="/hello.htm" class="springapp.web.HelloController"/>

</beans>
```

1.7. Copy libraries to `'WEB-INF/lib'`

First create a `'lib'` directory in the `'war/WEB-INF'` directory. Then, from the Spring distribution, copy `spring.jar` (from `spring-framework-2.5/dist`) and `spring-webmvc.jar` (from `spring-framework-2.5/dist/modules`) to the new `'war/WEB-INF/lib'` directory. Also, copy `commons-logging.jar` (from `spring-framework-2.5/lib/jakarta-commons`) to the `'war/WEB-INF/lib'` directory. These jars will be deployed to the server and they are also used during the build process.

1.8. Create the `Controller`

Create your `Controller` class – we are naming it `HelloController`, and it is defined in the `'springapp.web'` package. First we create the package directories and then we create the `'HelloController.java'` file and place it in the `'src/springapp/web'` directory.

`'springapp/src/springapp/web/HelloController.java'`:

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

import java.io.IOException;

public class HelloController implements Controller {

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

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

        logger.info("Returning hello view");

        return new ModelAndView("hello.jsp");
    }

}
```

This is a very basic `Controller` implementation. We will be expanding this later on as well as extending some of the base controller implementations provided by Spring. In Spring Web MVC, the `Controller` *handles* the request and returns a `ModelAndView` - in this case, one named `'hello.jsp'` which is also the name of the JSP file we will create next. The model that this class returns is actually resolved via a `ViewResolver`. Since we have not explicitly defined a `ViewResolver`, we are going to be given a default one by Spring that simply forwards to a URL matching the name of the view specified. We will modify this later on. We have also specified a logger so we can verify that we actually got into the handler. Using Tomcat, these log messages should show up in the `'catalina.out'` log file which can be found in the `'${appserver.home}/log'` directory of your Tomcat installation.

If you are using an IDE, you will want to configure your project's build path by adding the jars from the `'lib'` directory. You will also want to add `servlet-api.jar` from your servlet container's `'lib'` directory (`'${appserver.lib}'`). Adding these to your build path should successfully resolve all the import statements in the `'HelloController.java'` file.

1.9. Write a test for the `Controller`

Testing is a vital part of software development. It is also a core practice in Agile development. We have found that the best time to write tests is during development, not after, so even though our controller doesn't contain complex logic, we're going to write a test. This will allow us to make changes to it in the future with confidence. Let's create a new directory under `'springapp'` called `'test'`. This is where all our tests will go in a package structure that will mirror the source tree in `'springapp/src'`.

Create a test class called `'HelloControllerTests'` and make it extend JUnit's test class `TestCase`. It is a unit test that verifies the view name returned by `handleRequest()` matches the name of the view we expect: `'hello.jsp'`.

`'springapp/test/springapp/web/HelloControllerTests.java'`:

```java
package springapp.web;

import org.springframework.web.servlet.ModelAndView;

import springapp.web.HelloController;

import junit.framework.TestCase;

public class HelloControllerTests extends TestCase {

    public void testHandleRequestView() throws Exception{
        HelloController controller = new HelloController();
        ModelAndView modelAndView = controller.handleRequest(null, null);
        assertEquals("hello.jsp", modelAndView.getViewName());
    }
}
```

To run the test (and all the tests we're going to write), we need to add an Ant test task to our build script `'build.xml'`. First, we copy the `junit-3.8.2.jar` from `'spring-framework-2.5/lib/junit'` to `'war/WEB-INF/lib'`. Instead of creating a single task for compiling the tests and then running them, let's break them down into two distinct tasks: `'buildtests'` and `'tests'` which depends on `'buildtests'`.
If you are using an IDE, you may want to run your tests within your IDE. Configure your project's build path by adding the `junit-3.8.2.jar` to it.

`'springapp/build.xml'`:

```xml
    <property name="test.dir" value="test"/>

    <target name="buildtests" description="Compile test tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
            deprecation="false" optimize="false" failonerror="true">
            <src path="${test.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="master-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/*Tests.*"/>
                </fileset>
            </batchtest>
```

```
        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ***********************************************************
            ***********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ***********************************************************
            ***********************************************************
        </fail>
    </target>
```

Now run the Ant `'tests'` task and the test should pass.

```
$ ant tests
Buildfile: build.xml

build:

buildtests:
    [javac] Compiling 1 source file to /Users/Shared/Projects/springapp/war/WEB-INF/classes

tests:
    [junit] Running springapp.web.HelloWorldControllerTests
    [junit] Oct 30, 2007 11:31:43 PM springapp.web.HelloController handleRequest
    [junit] INFO: Returning hello view
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.03 sec
    [junit] Testsuite: springapp.web.HelloWorldControllerTests
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.03 sec

    [junit] ------------- Standard Error -----------------
    [junit] Oct 30, 2007 11:31:43 PM springapp.web.HelloController handleRequest
    [junit] INFO: Returning hello view
    [junit] ------------- --------------- ---------------

BUILD SUCCESSFUL
Total time: 2 seconds
```

Another of the best practices of Agile development is *Continuous Integration*. It's a good idea to ensure your tests are run with every build (ideally as automated project builds) so that you know your application logic is behaving as expected as the code evolves.

1.10. Create the `View`

Now it is time to create our first view. As we mentioned earlier, we are forwarding to a JSP page named `'hello.jsp'`. To begin with, we'll put it in the `'war'` directory.

`'springapp/war/hello.jsp'`:

```
<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
    <p>Greetings.</p>
  </body>
</html>
```

1.11. Compile and deploy the application

Run the `'deploy'` Ant target (which invokes the `'build'` target), and then run the `'reload'` task of the `'build.xml'` file. This will force a build and reload of the application in Tomcat. We have to check the Ant

output and the Tomcat logs for any possible deployment errors – such as typos in the above files or missing classes or jar files.

Here is a sample output from the Ant build:

```
$ ant deploy reload
Buildfile: build.xml

build:
    [mkdir] Created dir: /Users/trisberg/Projects/springapp/war/WEB-INF/classes
    [javac] Compiling 1 source file to /Users/trisberg/Projects/springapp/war/WEB-INF/classes

deploy:
     [copy] Copying 7 files to /Users/trisberg/apache-tomcat-5.5.17/webapps/springapp

BUILD SUCCESSFUL
Total time: 3 seconds
$ ant reload
Buildfile: build.xml

reload:
    [reload] OK - Reloaded application at context path /springapp

BUILD SUCCESSFUL
Total time: 2 seconds
```

And here is an excerpt from the Tomcat `'catalina.out'` log file.

```
Oct 30, 2007 11:43:09 PM org.springframework.web.servlet.FrameworkServlet initServletBean
INFO: FrameworkServlet 'springapp': initialization started
Oct 30, 2007 11:43:09 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.XmlWebApplicationContext@6576d5: display name
[WebApplicationContext for namespace 'springapp-servlet']; startup date [Tue Oct 30 23:43:09 GMT 2007];
...
...
Oct 30, 2007 11:43:09 PM org.springframework.web.servlet.FrameworkServlet initServletBean
INFO: FrameworkServlet 'springapp': initialization completed in 150 ms
```

1.12. Try out the application

Let's try this new version of the application.

Open a browser and browse to http://localhost:8080/springapp/hello.htm.
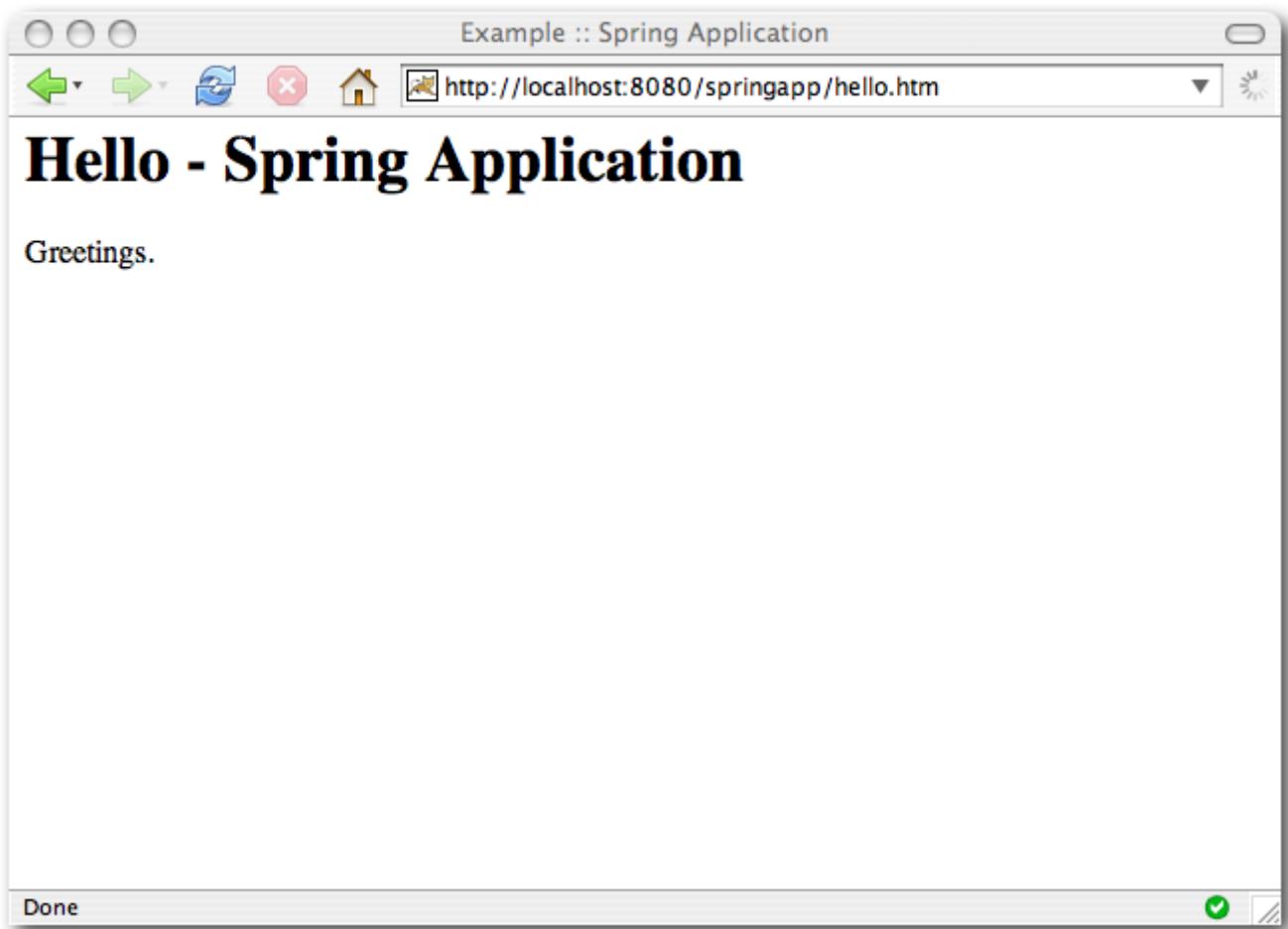
The updated application

1.13. Summary

Let's take quick look at the parts of our application that we have created so far.

1. An introduction page, `'index.jsp'`, the welcome page of the application. It was used to test our setup was correct. We will later change this to actually provide a link into our application.

2. A `DispatcherServlet` (front controller) with a corresponding `'springapp-servlet.xml'` configuration file.

3. A page controller, `HelloController`, with limited functionality – it just returns a `ModelAndView`. We currently have an empty model and will be providing a full model later on.

4. A unit test class for the page controller, `HelloControllerTests`, to verify the name of the view is the one we expect.

5. A view, `'hello.jsp'`, that again is extremely basic. The good news is the whole setup works and we are now ready to add more functionality.

Find below a screen shot of what your project directory structure must look like after following the above instructions.
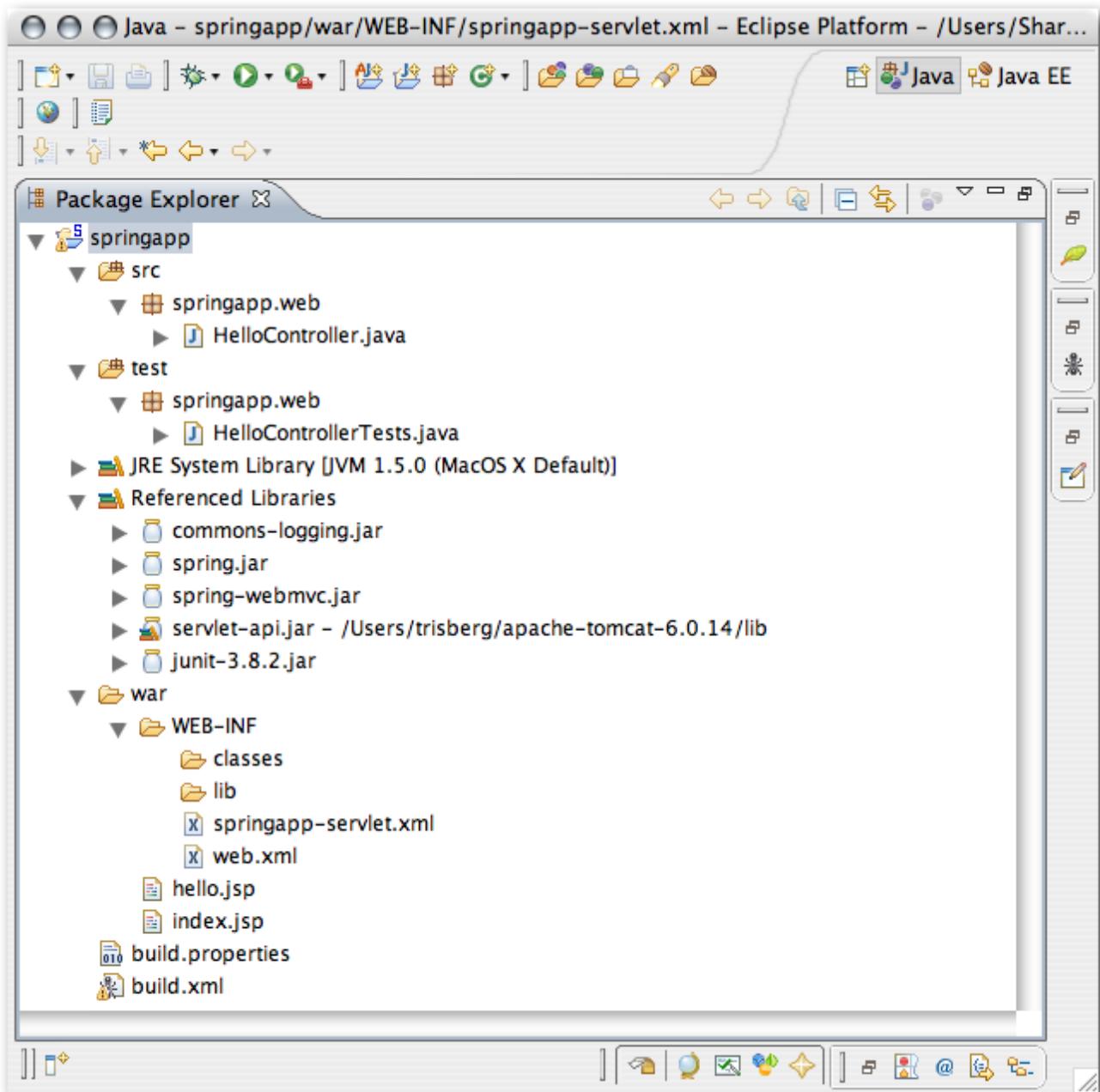
The project directory structure at the end of part 1

Chapter 2. Developing and Configuring the Views and the Controller

This is Part 2 of a step-by-step tutorial on 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 that we will now flesh out.

This is what we have implemented so far:

- An introduction page, `'index.jsp'`, the welcome page of the application. It was used to test our setup was correct. We will later change this to actually provide a link into our application.

- A `DispatcherServlet` (front controller) with a corresponding `'springapp-servlet.xml'` configuration file.

- A page controller, `HelloController`, with limited functionality – it just returns a `ModelAndView`. We currently have an empty model and will be providing a full model later on.

- A unit test class for the page controller, `HelloControllerTests`, to verify the name of the view is the one we expect.

- A view, `'hello.jsp'`, that again is extremely basic. The good news is the whole setup works and we are now ready to add more functionality.

2.1. Configure JSTL and add JSP header file

We will be using the JSP Standard Tag Library (JSTL), so let's start by copying the JSTL files we need to our `'WEB-INF/lib'` directory. Copy `jstl.jar` from the `'spring-framework-2.5/lib/j2ee'` directory and `standard.jar` from the `'spring-framework-2.5/lib/jakarta-taglibs'` directory to the `'springapp/war/WEB-INF/lib'` directory.

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 `'jsp'` under the `'WEB-INF'` directory. This will ensure that views can only be accessed via the controller since it will not be not 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 `'jsp'` directory up a level. You would then use `'springapp/war/jsp'` as the directory instead of `'springapp/war/WEB-INF/jsp'` in all the code examples that will follow.

First we create the header file for inclusion in all the JSPs we create.

`'springapp/war/WEB-INF/jsp/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" %>
```

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:

`'springapp/war/index.jsp'`:

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

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

Move `'hello.jsp'` to the `'WEB-INF/jsp'` directory. Add the same include directive we added to `'index.jsp'` to `'hello.jsp'`. We also add the current date and time as output to be retrieved from the model passed to the view which will be rendered using the JSTL <c:out/> tag.

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

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

2.2. Improve the controller

Before we update the location of the JSP in our controller, let's update our unit test class first. We know we need to update the view's resource reference with its new location `'WEB-INF/jsp/hello.jsp'`. We also know there should be an object in the model mapped to the key `"now"`.

`'springapp/tests/HelloControllerTests.java'`:

```
package springapp.web;

import org.springframework.web.servlet.ModelAndView;

import springapp.web.HelloController;

import junit.framework.TestCase;

public class HelloControllerTests extends TestCase {

    public void testHandleRequestView() throws Exception{
        HelloController controller = new HelloController();
        ModelAndView modelAndView = controller.handleRequest(null, null);
        assertEquals("WEB-INF/jsp/hello.jsp", modelAndView.getViewName());
        assertNotNull(modelAndView.getModel());
        String nowValue = (String) modelAndView.getModel().get("now");
        assertNotNull(nowValue);
    }
}
```

Next, we run the Ant `'tests'` target and our test should fail.

```
$ ant tests
Buildfile: build.xml

build:

buildtests:
    [javac] Compiling 1 source file to /home/trisberg/workspace/springapp/war/WEB-INF/classes

tests:
    [junit] Running springapp.web.HelloControllerTests
    [junit] Testsuite: springapp.web.HelloControllerTests
    [junit] Oct 31, 2007 1:27:10 PM springapp.web.HelloController handleRequest
```

```
    [junit] INFO: Returning hello view
    [junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.046 sec
    [junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.046 sec
    [junit]
    [junit] ------------- Standard Error -----------------
    [junit] Oct 31, 2007 1:27:10 PM springapp.web.HelloController handleRequest
    [junit] INFO: Returning hello view
    [junit] ------------- --------------- ---------------
    [junit] Testcase: testHandleRequestView(springapp.web.HelloControllerTests):      FAILED
    [junit] expected:<[WEB-INF/jsp/]hello.jsp> but was:<[]hello.jsp>
    [junit] junit.framework.ComparisonFailure: expected:<[WEB-INF/jsp/]hello.jsp> but was:<[]hello.jsp>
    [junit]     at springapp.web.HelloControllerTests.testHandleRequestView(HelloControllerTests.java:14)
    [junit]
    [junit]
    [junit] Test springapp.web.HelloControllerTests FAILED

BUILD FAILED
/home/trisberg/workspace/springapp/build.xml:101: tests.failed=true
            ************************************************************
            ************************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ************************************************************
            ************************************************************

Total time: 2 seconds
```

Now we update `HelloController` by setting the view's resource reference to its new location `'WEB-INF/jsp/hello.jsp'` as well as set the key/value pair for the current date and time value in the model with the key identifier: `"now"` and the string value: `'now'`.

`'springapp/src/springapp/web/HelloController.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 org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.io.IOException;
import java.util.Date;

public class HelloController implements Controller {

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

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

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

        return new ModelAndView("WEB-INF/jsp/hello.jsp", "now", now);
    }

}
```

We rerun our `'tests'` target and the test passes.

Remember that the `Controller` has already been configured in `'springapp-servlet.xml'` file, so we are ready to try out our enhancements after we build and deploy this new code. When we enter http://localhost:8080/springapp/ in a browser, it should pull up the welcome file `'index.jsp'`, which should redirect to `'hello.htm'` and is handled by the `DispatcherServlet`, which in turn delegates our request to the page controller that puts the date and time in the model and then makes the model available to the view

`'hello.jsp'`.
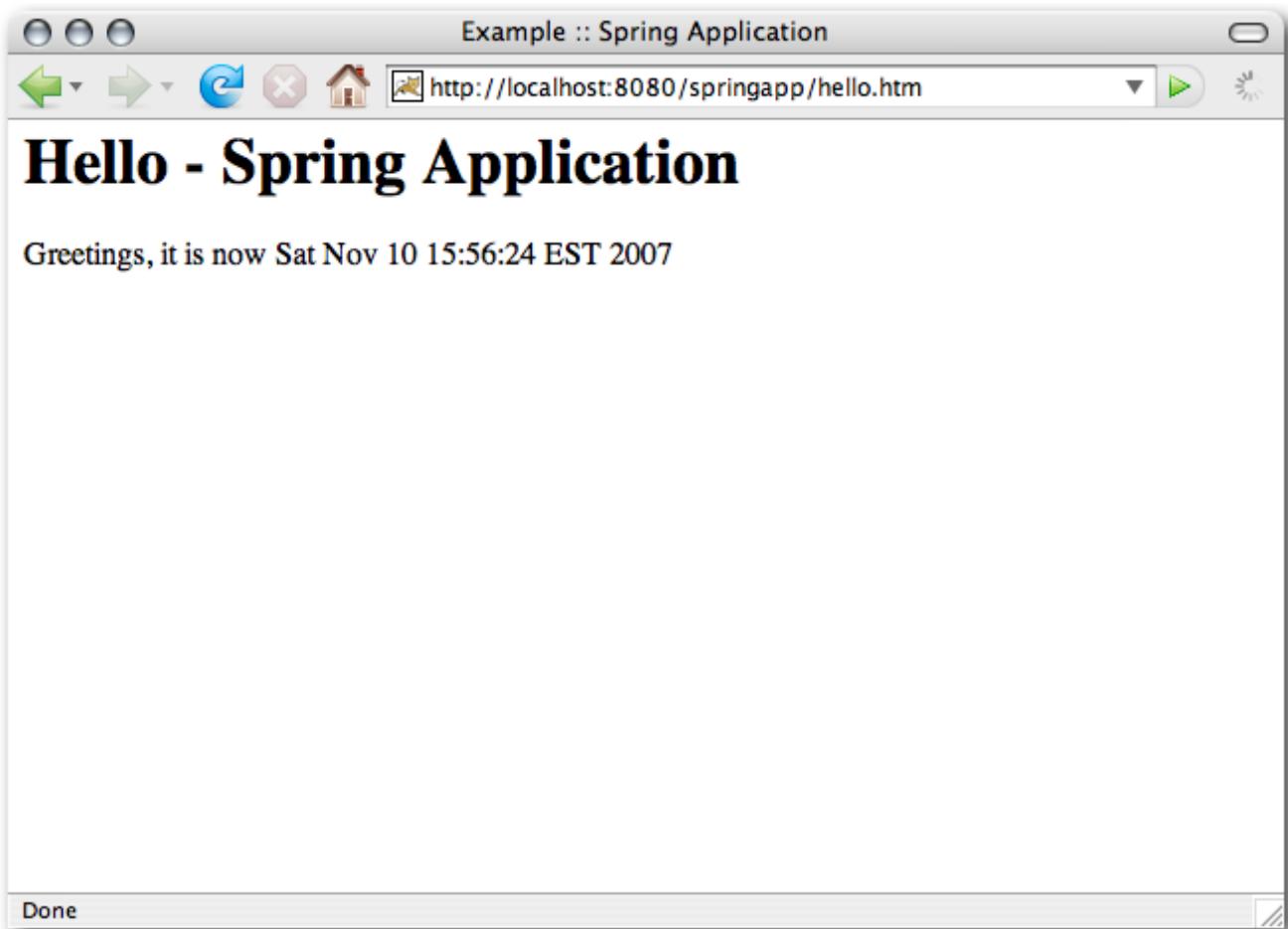
The updated application

2.3. Decouple the view from the controller

Right now the controller specifies the full path of the view, which creates an unnecessary dependency between the controller and the view. Ideally we would like to map to the view using a logical name, allowing us to switch the view without having to change the controller. You can set this mapping in a properties file if you like using a `ResourceBundleViewResolver` and a `SimpleUrlHandlerMapping` class. For the basic mapping of a view to a location, simply set a prefix and a suffix on the `InternalResourceViewResolver`. This second approach is the one that we will implement now, so we modify the `'springapp-servlet.xml'` and declare a `'viewResolver'` entry. By choosing the `JstlView`, it will enable us to use JSTL in combination with message resource bundles as well as provide us with the support for internationalization.

`'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-2.0.xsd">

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

    <bean name="/hello.htm" class="springapp.web.HelloController"/>

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

```
    </bean>

</beans>
```

We update the view name in the controller test class `HelloControllerTests` to `'hello'` and rerun the test to check it fails.

`'springapp/test/springapp/web/HelloControllerTests.java'`:

```
package springapp.web;

import org.springframework.web.servlet.ModelAndView;

import springapp.web.HelloController;

import junit.framework.TestCase;

public class HelloControllerTests extends TestCase {

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

We then remove the prefix and suffix from the view name in the controller, leaving the controller to reference the view by its logical name `"hello"`.

`'springapp/src/springapp/web/HelloController.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 org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.io.IOException;
import java.util.Date;

public class HelloController implements Controller {

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

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

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

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

}
```

Rerun the test and it should now pass.

Let's compile and deploy the application and verify the application still works.

2.4. Summary

Let's take quick look at what we have created in Part 2.

1. A header file `'include.jsp'`, the JSP that contains the taglib directives for the tag libraries we'll be using in our JSPs.

These are the existing artifacts we have changed in Part 2.

1. The `HelloControllerTests` has been updated repeatedly as we make the controller reference the logical name of a view instead of its hard coded name and location.

2. The page controller, `HelloController`, now references the view by its logical view name through the use of the `'InternalResourceViewResolver'` defined in `'springapp-servlet.xml'`.

Find below a screen shot of what your project directory structure must look like after following the above instructions.
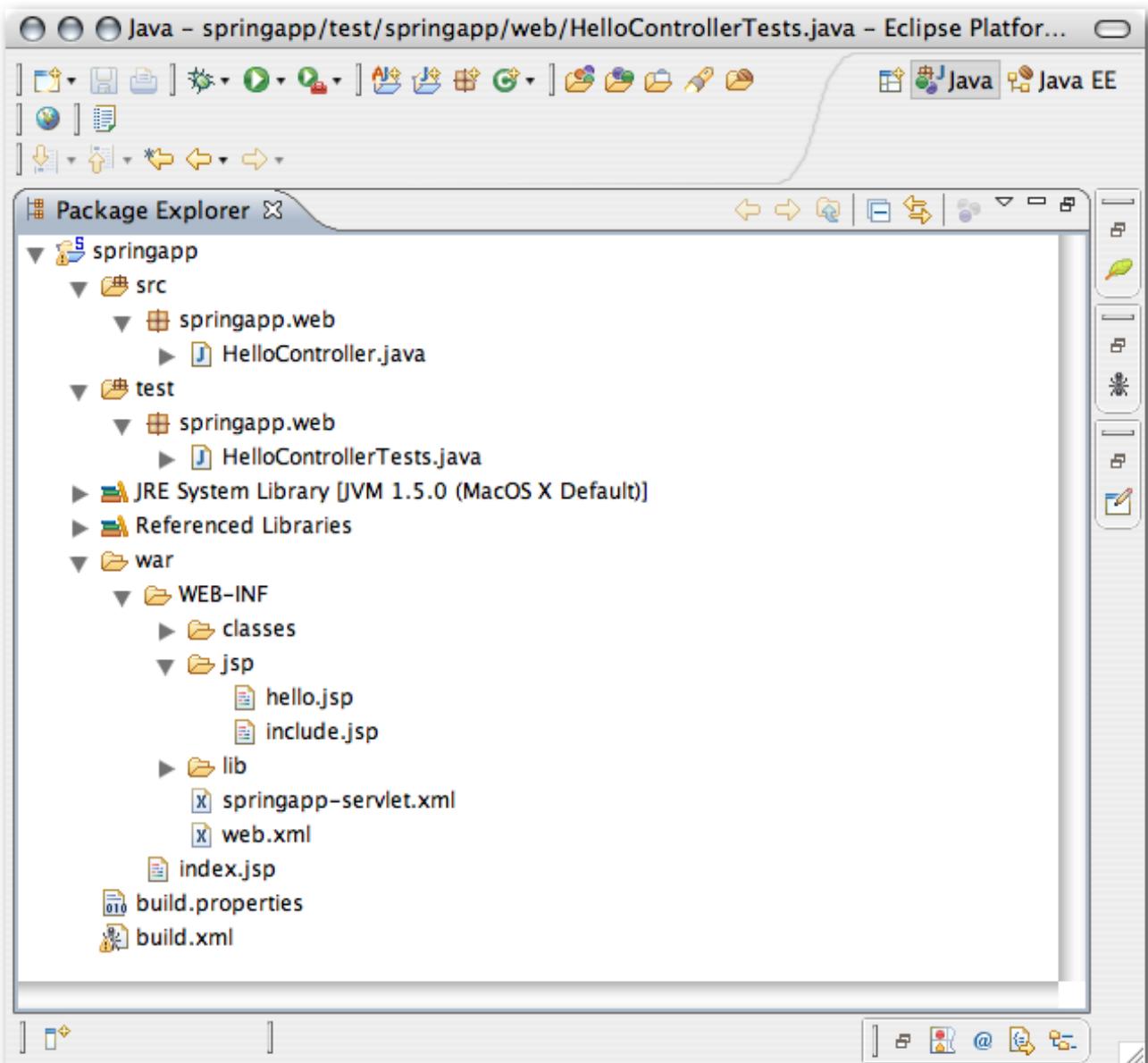
The project directory structure at the end of part 2

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.

3.1. 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.
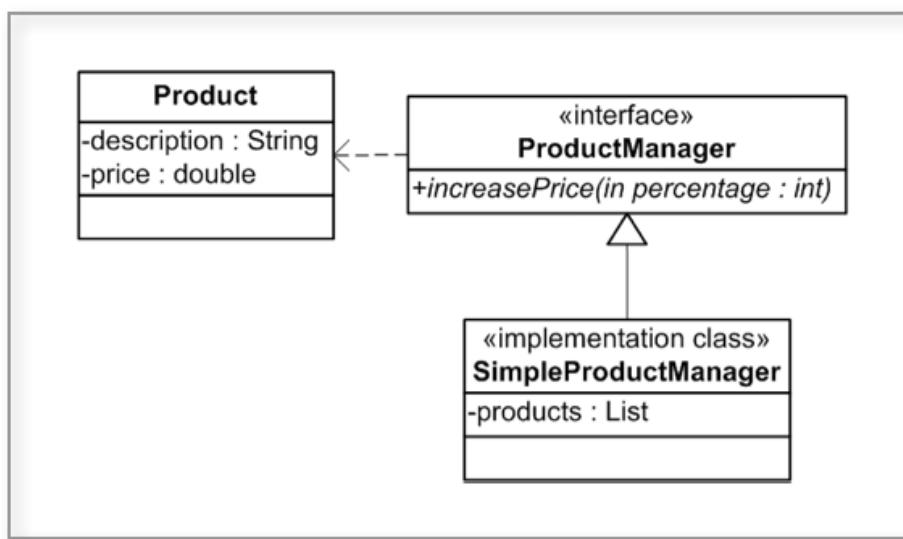
The class diagram for the inventory management system

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. In order to separate the web dependent logic from the business logic, we will place classes related to the web tier in the `'web'` package and create two new packages: one for service objects called `'service'` and another for domain objects called `'domain'`.

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/springapp/domain/Product.java'`:

```
package springapp.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.

`'springapp/test/springapp/domain/ProductTests.java'`:

```
package springapp.domain;

import junit.framework.TestCase;

public class ProductTests extends TestCase {

    private Product product;

    protected void setUp() throws Exception {
        product = new Product();
    }
```

```
    public void testSetAndGetDescription() {
        String testDescription = "aDescription";
        assertNull(product.getDescription());
        product.setDescription(testDescription);
        assertEquals(testDescription, product.getDescription());
    }

    public void testSetAndGetPrice() {
        double testPrice = 100.00;
        assertEquals(0, 0, 0);
        product.setPrice(testPrice);
        assertEquals(testPrice, product.getPrice(), 0);
    }

}
```

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).

`'springapp/src/springapp/service/ProductManager.java'`:

```
package springapp.service;

import java.io.Serializable;
import java.util.List;

import springapp.domain.Product;

public interface ProductManager extends Serializable{

    public void increasePrice(int percentage);

    public List<Product> getProducts();

}
```

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

`'springapp/src/springapp/service/SimpleProductManager.java'`:

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    public List<Product> getProducts() {
        throw new UnsupportedOperationException();
    }

    public void increasePrice(int percentage) {
        throw new UnsupportedOperationException();
    }

    public void setProducts(List<Product> products) {
        throw new UnsupportedOperationException();
    }

}
```

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`.

`'springapp/test/springapp/service/SimpleProductManagerTests.java'`:

```java
package springapp.service;

import junit.framework.TestCase;

public class SimpleProductManagerTests extends TestCase {

    private SimpleProductManager productManager;

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
    }

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

}
```

Rerun all the Ant `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.

`'springapp/test/springapp/service/SimpleProductManagerTests.java'`:

```java
package springapp.service;

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

import springapp.domain.Product;

import junit.framework.TestCase;

public class SimpleProductManagerTests 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 Ant `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.

`'springapp/src/springapp/service/SimpleProductManager.java'`:

```
package springapp.service;

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

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    private List<Product> products;

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

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

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

}
```

Rerun the Ant `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.

`'springapp/test/springapp/service/SimpleProductManagerTests.java'`:

```
package springapp.service;

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

import springapp.domain.Product;

import junit.framework.TestCase;

public class SimpleProductManagerTests 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()`.

`'springapp/src/springapp/service/SimpleProductManager.java'`:

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

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;
    }

}
```

Rerun the Ant `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.

3.3. Summary

Let's take quick look at what we did in Part 3.

1. We implemented a domain object `Product` and a service interface `ProductManager` and concrete class `SimpleProductManager` all as POJOs.

2. We wrote unit tests for all the classes we implemented.

3. We didn't write a line of code to do with Spring. This is an example of how non-invasive the Spring Framework really is. One of its core aims is to enable developers to focus on tackling the most important task of all: to deliver value by modelling and implementing business requirements. Another of its aims is to make following best practices easy, such as implementing services using interfaces and unit testing as much as is pragmatic given project constraints. Over the course of this tutorial, you'll see the benefits of designing to interfaces come to life.

Find below a screen shot of what your project directory structure must look like after following the above instructions.
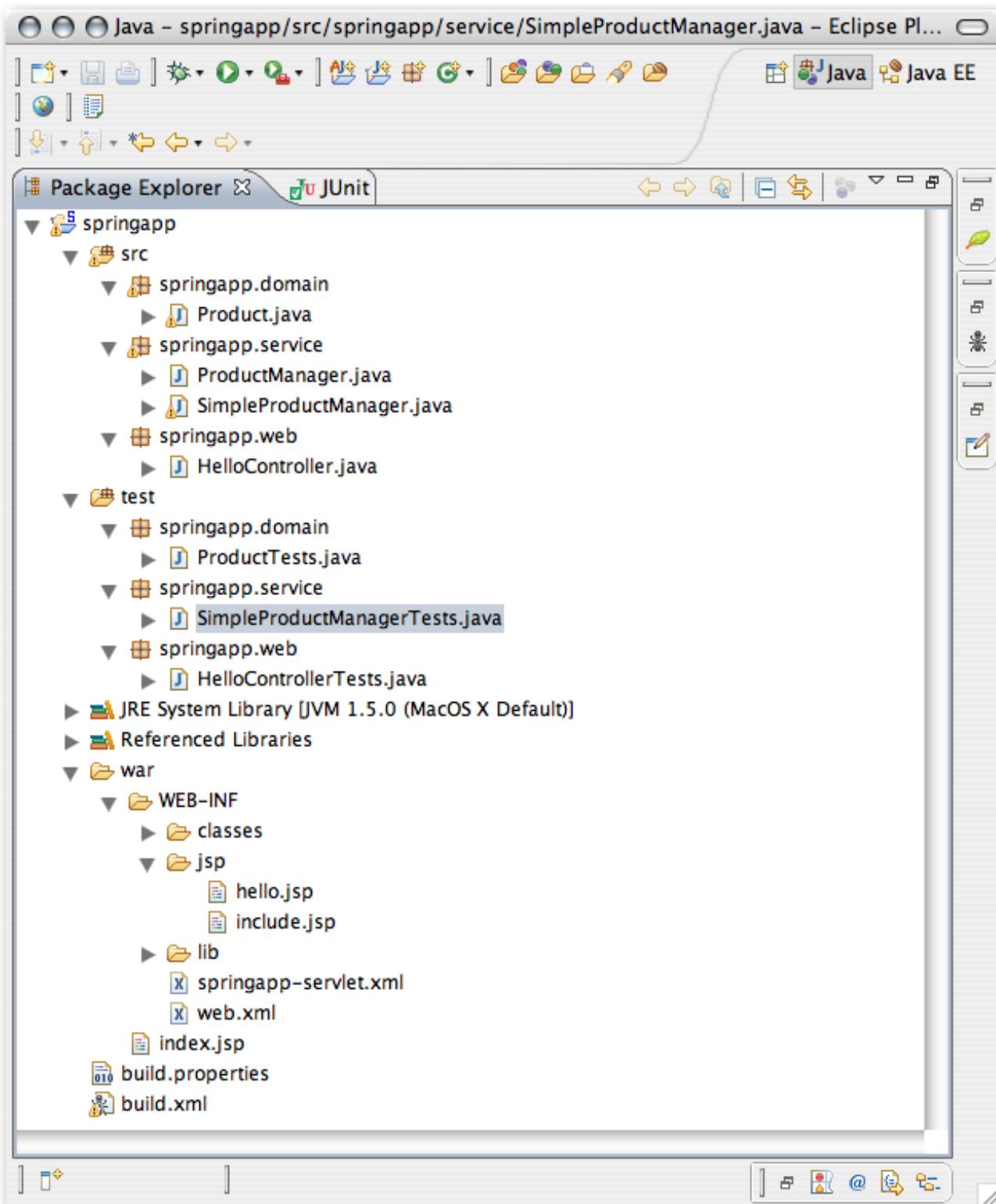
The project directory structure at the end of part 3

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'`:

```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'`:

```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'`:

```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
<?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':

```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:
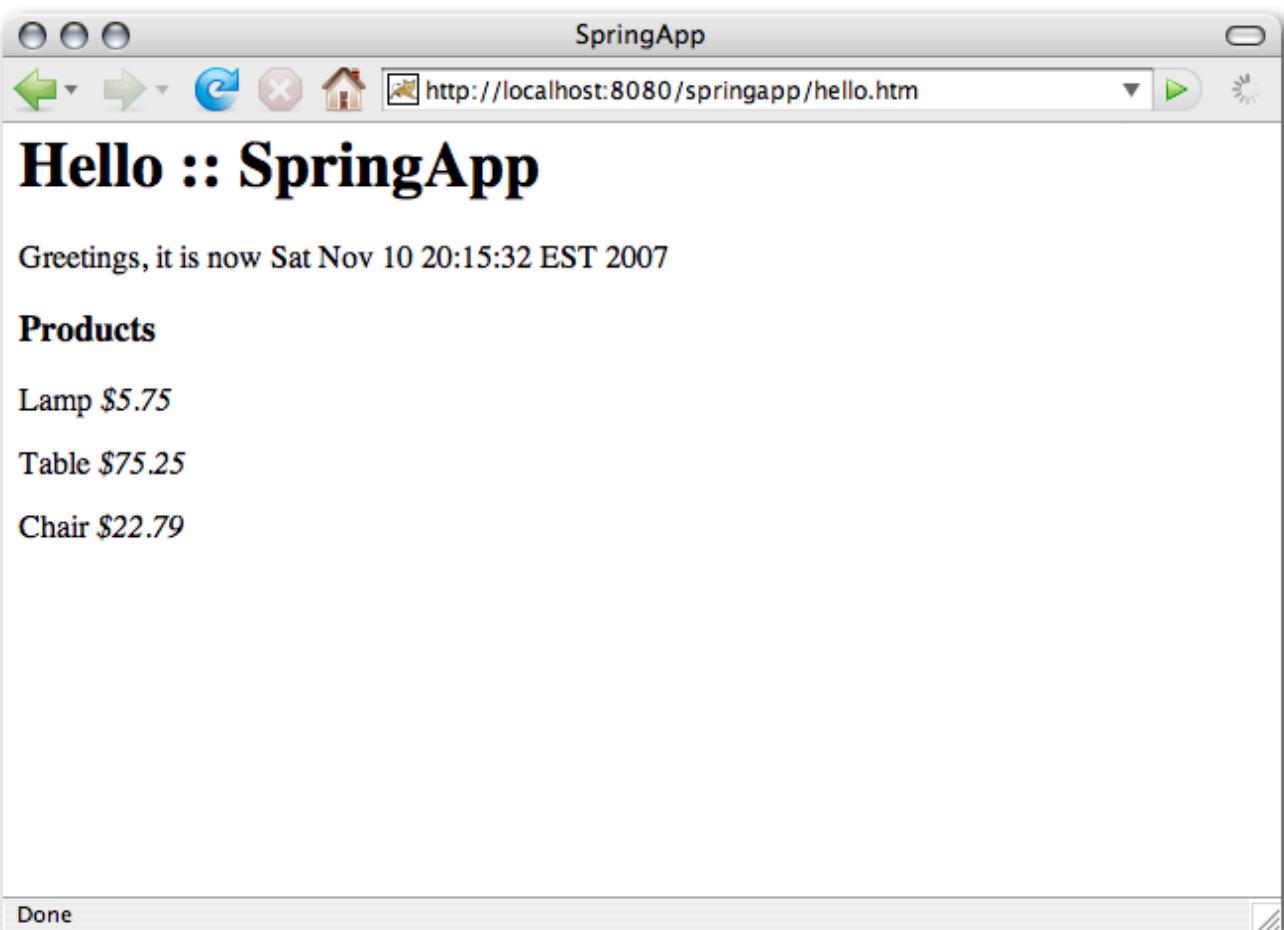
The updated application

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
<?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':

```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
<?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'`:

```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.
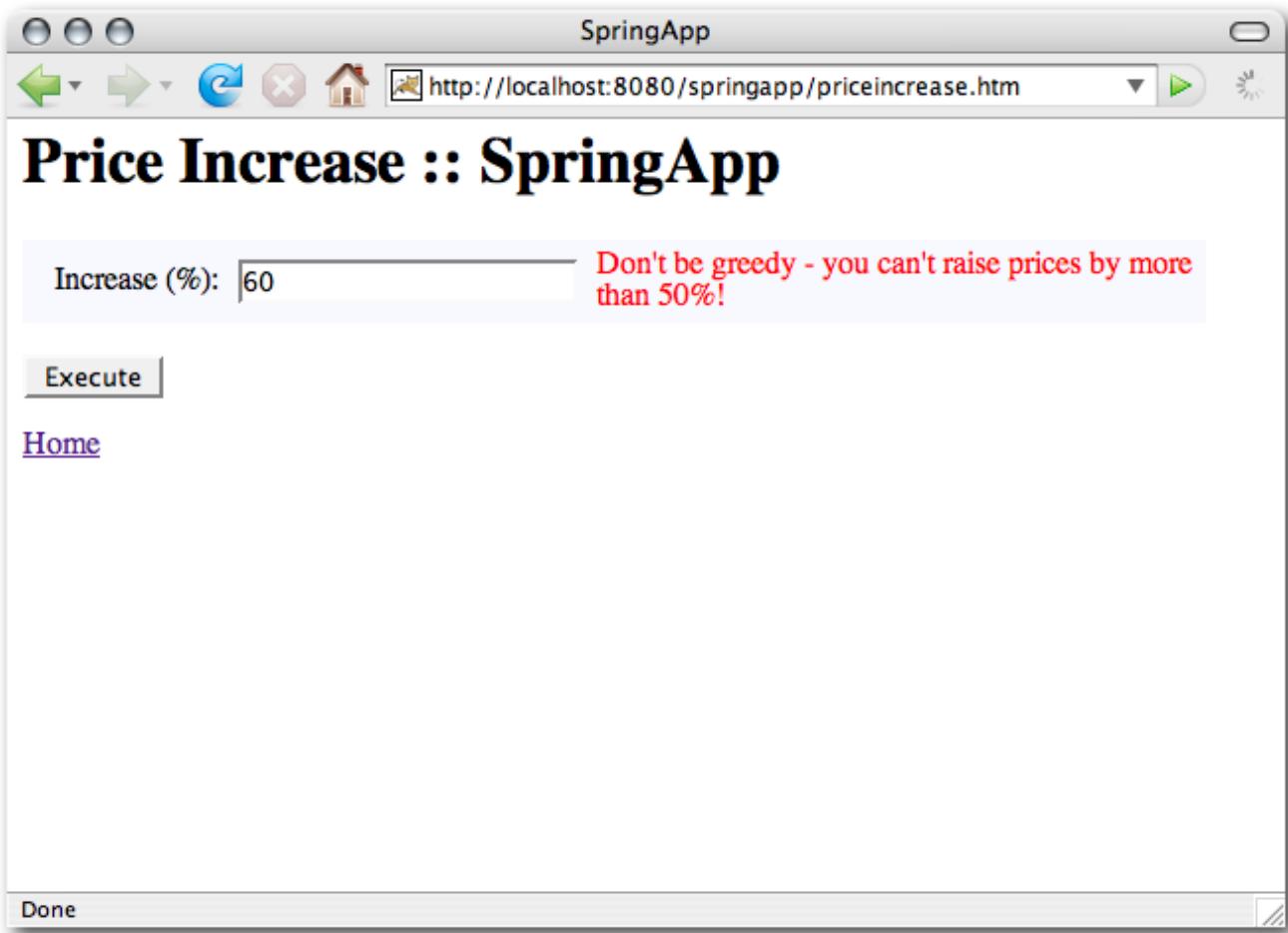
The updated application

4.7. Summary

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

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

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.
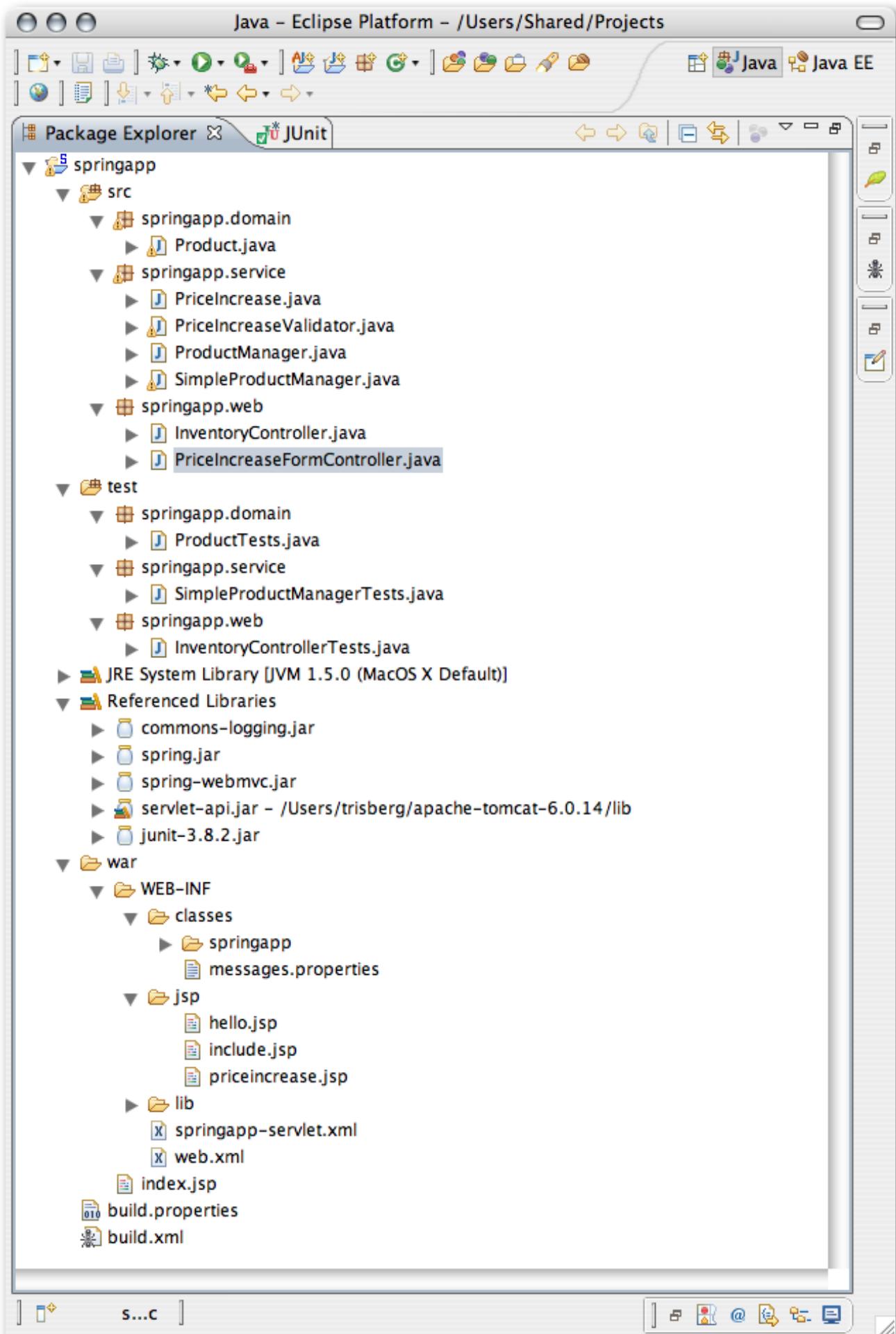
Chapter 5. Implementing Database Persistence

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. Create database startup script

Before we can start developing the persistence code, we need a database. We are planning on using HSQL, which is a good open source database written in Java. This database is distributed with Spring, so we can just copy the jar file to the web application's lib directory. Copy `hsqldb.jar` from the `'spring-framework-2.5/lib/hsqldb'` directory to the `'springapp/war/WEB-INF/lib'` directory. We will use HSQL in standalone server mode. That means we will have to start up a separate database server instead of relying on an embedded database, but it gives us easier access to see changes made to the database when running the web application.

We need a script or batch file to start the database. Create a `'db'` directory under the main `'springapp'` directory. This new directory will contain the database files. Now, let's and add a startup script:

For Linux/Mac OS X add:

`'springapp/db/server.sh'`:

```
java -classpath ../war/WEB-INF/lib/hsqldb.jar org.hsqldb.Server -database test
```

Don't forget to change the execute permission by running **'chmod +x server.sh'**.

For Windows add:

`'springapp/db/server.bat'`:

```
java -classpath ..\war\WEB-INF\lib\hsqldb.jar org.hsqldb.Server -database test
```

Now you can open a command window, change to the `'springapp/db'` directory and start the database by running one of these startup scripts.

5.2. Create table and test data scripts

First, lets review the SQL statement needed to create the table. We create the file 'create_products.sql' in the db directory.

`'springapp/db/create_products.sql'`:

```
CREATE TABLE products (
  id INTEGER NOT NULL PRIMARY KEY,
  description varchar(255),
  price decimal(15,2)
);
```

```
CREATE INDEX products_description ON products(description);
```

Now we need to add our test data. Create the file 'load_data.sql' in the db directory.

`'springapp/db/load_data.sql'`:

```
INSERT INTO products (id, description, price) values(1, 'Lamp', 5.78);
INSERT INTO products (id, description, price) values(2, 'Table', 75.29);
INSERT INTO products (id, description, price) values(3, 'Chair', 22.81);
```

In the following section we will add some Ant targets to the build script so that we can run these SQL scripts.

5.3. Add Ant tasks to run scripts and load test data

We will create tables and populate them with test data using Ant's built-in "sql" task. To use this we need to add some database connection properties to the build properties file.

`'springapp/build.properties'`:

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret

db.driver=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:hsql://localhost
db.user=sa
db.pw=
```

Next we add the targets we need to the build script. There are targets to create and delete tables and to load and delete test data.

Add the following targets to `'springapp/build.xml'`:

```
    <target name="createTables">
        <echo message="CREATE TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue"
            src="db/create_products.sql">
          <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="dropTables">
        <echo message="DROP TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue">
          <classpath refid="master-classpath"/>

        DROP TABLE products;
```

```
            </sql>
    </target>

    <target name="loadData">
        <echo message="LOAD DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/load_data.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="printData">
        <echo message="PRINT DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             print="true">
            <classpath refid="master-classpath"/>

        SELECT * FROM products;

        </sql>
    </target>

    <target name="clearData">
        <echo message="CLEAR DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        DELETE FROM products;

        </sql>
    </target>

    <target name="shutdownDb">
        <echo message="SHUT DOWN DATABASE USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        SHUTDOWN;

        </sql>
    </target>
```

Now you can execute **'ant createTables loadData printData'** to prepare the test data we will use later.

5.4. Create a Data Access Object (DAO) implementation for JDBC

Begin with creating a new `'springapp/src/repository'` directory to contain any classes that are used for database access. In this directory we create a new interface called `ProductDao`. This will be the interface that defines the functionality that the DAO implementation classes will provide – we could choose to have more than one implementation some day.

'springapp/src/springapp/repository/ProductDao.java':

```
package springapp.repository;

import java.util.List;

import springapp.domain.Product;

public interface ProductDao {

    public List<Product> getProductList();

    public void saveProduct(Product prod);

}
```

We'll follow this with a class called `JdbcProductDao` that will be the JDBC implementation of this interface. Spring provides a JDBC abstraction framework that we will make use of. The biggest difference between using straight JDBC and Spring's JDBC framework is that you don't have to worry about opening and closing the connection or any statements. It is all handled for you. Another advantage is that you won't have to catch any exceptions, unless you want to. Spring wraps all SQLExceptions in it's own unchecked exception hierarchy inheriting from DataAccessException. If you want to, you can catch this exception, but since most database exceptions are impossible to recover from anyway, you might as well just let the exception propagate up to a higher level. The class `SimpleJdbcDaoSupport` provides convenient access to an already configured `SimpleJdbcTemplate`, so we extend this class. All we will have to provide in the application context is a configured `DataSource`.

'springapp/src/springapp/repository/JdbcProductDao.java':

```
package springapp.repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

import springapp.domain.Product;

public class JdbcProductDao extends SimpleJdbcDaoSupport implements ProductDao {

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


    public List<Product> getProductList() {
        logger.info("Getting products!");
        List<Product> products = getSimpleJdbcTemplate().query(
                "select id, description, price from products",
                new ProductMapper());
        return products;
    }

    public void saveProduct(Product prod) {
        logger.info("Saving product: " + prod.getDescription());
        int count = getSimpleJdbcTemplate().update(
            "update products set description = :description, price = :price where id = :id",
            new MapSqlParameterSource().addValue("description", prod.getDescription())
                .addValue("price", prod.getPrice())
                .addValue("id", prod.getId()));
        logger.info("Rows affected: " + count);
    }

    private static class ProductMapper implements ParameterizedRowMapper<Product> {

        public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
```

```
        Product prod = new Product();
        prod.setId(rs.getInt("id"));
        prod.setDescription(rs.getString("description"));
        prod.setPrice(new Double(rs.getDouble("price")));
        return prod;
    }

  }

}
```

Let's go over the two DAO methods in this class. Since we are extending `SimpleJdbcSupport` we get a `SimpleJdbcTemplate` prepared and ready to use. This is accessed by calling the `getSimpleJdbcTemplate()` method.

The first method, `getProductList()` executes a query using the `SimpleJdbcTemplate`. We simply provide the SQL statement and a class that can handle the mapping between the `ResultSet` and the `Product` class. In our case the row mapper is a class named `ProductMapper` that we define as an inner class of the DAO. This class will so far not be used outside of the DAO so making it an inner class works well.

The ProductMapper implements the `ParameterizedRowMapper` interface that defines a single method named `mapRow` that must be implemented. This method will map the data from each row into a class that represents the entity you are retrieving in your query. Since the `RowMapper` is parameterized, the `mapRow` method returns the actual type that is created.

The second method saveProduct is also using the `SimplJdbcTemplate`. This time we are calling the `update` method passing in an SQL statement together with the parameter values in the form of a `MapSqlParameterSource`. Using a `MapSqlParameterSource` allows us to use named parameters instead of the typical "?" place holders that you are used to from writing plain JDBC. The named parameters makes your code more explicit and you avoid problems caused by parameters being set out of order etc. The update method returns the count of rows affected.

We need to store the value of the primary key for each product in the Product class. This key will be used when we persist any changes to the object back to the database. To hold this key we add a private field named 'id' complete with setters and getters to Product.java.

`'springapp/src/springapp/domain/Product.java'`:

```java
package springapp.domain;

import java.io.Serializable;

public class Product implements Serializable {

    private int id;
    private String description;
    private Double price;

    public void setId(int i) {
        id = i;
    }

    public int getId() {
        return 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();
    }
}
```

This completes the Simple JDBC implementation of our persistence layer.

5.5. Implement tests for JDBC DAO implementation

Time to add tests for the JDBC DAO implementation. Spring provides an extensive testing framework that supports JUnit 3.8 and 4 as well as TestNG. We can't cover all of that in this guide but we will show a simple implementation of the JUnit 3.8 specific support. We need to add the jar file containing the Spring test framework to our project. Copy `spring-test.jar` from the `'spring-framework-2.5/dist/modules'` directory to the `'springapp/war/WEB-INF/lib'` directory.

Now we can create our test class. By extending `AbstractTransactionalDataSourceSpringContextTests` we get a lot of nice features for free. We get dependency injection of any public setter from an application context. This application context is loaded by the test framework. All we need to do is specify the name for it in the `getConfigLocations` method. We also get an opportunity to prepare our database with the appropriate test data in the `onSetUpInTransaction` method. This is important, since we don't know the state of the database when we run our tests. As long as the table exists we will clear it and load what we need for our tests. Since we are extending a "Transactional" test, any changes we make will be automatically rolled back once the test finishes. The `deleteFromTables` and `executeSqlScript` methods are defined in the super class, so we don't have to implement them for each test. Just pass in the table names to be cleared and the name of the script that contains the test data.

`'springapp/test/springapp/domain/JdbcProductDaoTests.java'`:

```
package springapp.repository;

import java.util.List;
import org.springframework.test.AbstractTransactionalDataSourceSpringContextTests;
import springapp.domain.Product;

public class JdbcProductDaoTests extends AbstractTransactionalDataSourceSpringContextTests {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Override
    protected String[] getConfigLocations() {
        return new String[] {"classpath:test-context.xml"};
    }

    @Override
    protected void onSetUpInTransaction() throws Exception {
        super.deleteFromTables(new String[] {"products"});
        super.executeSqlScript("file:db/load_data.sql", true);
    }

    public void testGetProductList() {
        List<Product> products = productDao.getProductList();
```

```
        assertEquals("wrong number of products?", 3, products.size());
    }

    public void testSaveProduct() {
        List<Product> products = productDao.getProductList();
        for (Product p : products) {
            p.setPrice(200.12);
            productDao.saveProduct(p);
        }
        List<Product> updatedProducts = productDao.getProductList();
        for (Product p : updatedProducts) {
            assertEquals("wrong price of product?", 200.12, p.getPrice());
        }
    }
}
```

We don't have the application context file that is loaded for this test yet, so let's create this file in the `'springapp/test'` directory:

`'springapp/test/test-context.xml'`:

```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 test application context definition for the jdbc based tests -->

    <bean id="productDao" class="springapp.repository.JdbcProductDao">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="${jdbc.driverClassName}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username"  value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
    </bean>
    <bean id="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

We have defined a `productDao` which is the class we are testing. We have also defined a `DataSource` with place holders for the configuration values. These values are provided via a separate property file and at runtime, the `PropertyPlaceholderConfigurer` that we have defined will read this property file and substitute the place holders with the actual values. This is convenient since this isolates the connection values into their own file. These values often need to be changed during application deployment. We put this new file in the `'war/WEB-INF/classes'` directory so it will be available when we run the application and also later when we deploy the web application. The content of this property file is:

`'springapp/war/WEB-INF/classes/jdbc.properties'`:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost
jdbc.username=sa
```

```
jdbc.password=
```

Since we added a configuration file to the `'test'`" directory and a jdbc.properties file to the `'WEB-INF/classes'` directory, let's add a new classpath entry for our tests. It shoud go after the definition of the `'test.dir'` property:

`'springapp/build.xml'`:

```
...
    <property name="test.dir" value="test"/>


    <path id="test-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
        <pathelement path="${test.dir}"/>
        <pathelement path="${web.dir}/WEB-INF/classes"/>
    </path>

...
```

We should now have enough for our tests to run and pass but we want to make one additional change to the build script. It's a good practice to separate any integration tests that depend on a live database from the rest of the tests. So we add a separate "dbTests" target to our build script and exclude the database tests from the "tests" target.

`'springapp/build.xml'`:

```
...

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/*Tests.*"/>
                    <exclude name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            *********************************************************
            *********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            *********************************************************
            *********************************************************
        </fail>
    </target>

    <target name="dbTests" depends="build, buildtests,dropTables,createTables,loadData"
            description="Run db tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>
```

```
            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ************************************************************
            ************************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ************************************************************
            ************************************************************
        </fail>
    </target>

...
```

Time to run this test, execute **'ant dbTests'** to see if the tests pass.

5.6. Summary

We have now completed the persistence layer and in the next part we will integrate it with our web application. But first, lets quickly summarize hat we accomplished in this part.

1. First we configured our database and created start-up scripts.

2. We created scripts to use when creating the table and also to load some test data.

3. Next we added some tasks to our build script to run when we needed to create or delete the table and also when we needed to add test data or delete the data.

4. We created the actual DAO class that will handle the persistence work using Spring's `SimpeJdbcTemplate`.

5. Finally we created unit or more accurately integration tests and corresponding ant targets to run these tests.

Below is a screen shot of what your project directory structure should look like after following the above instructions.
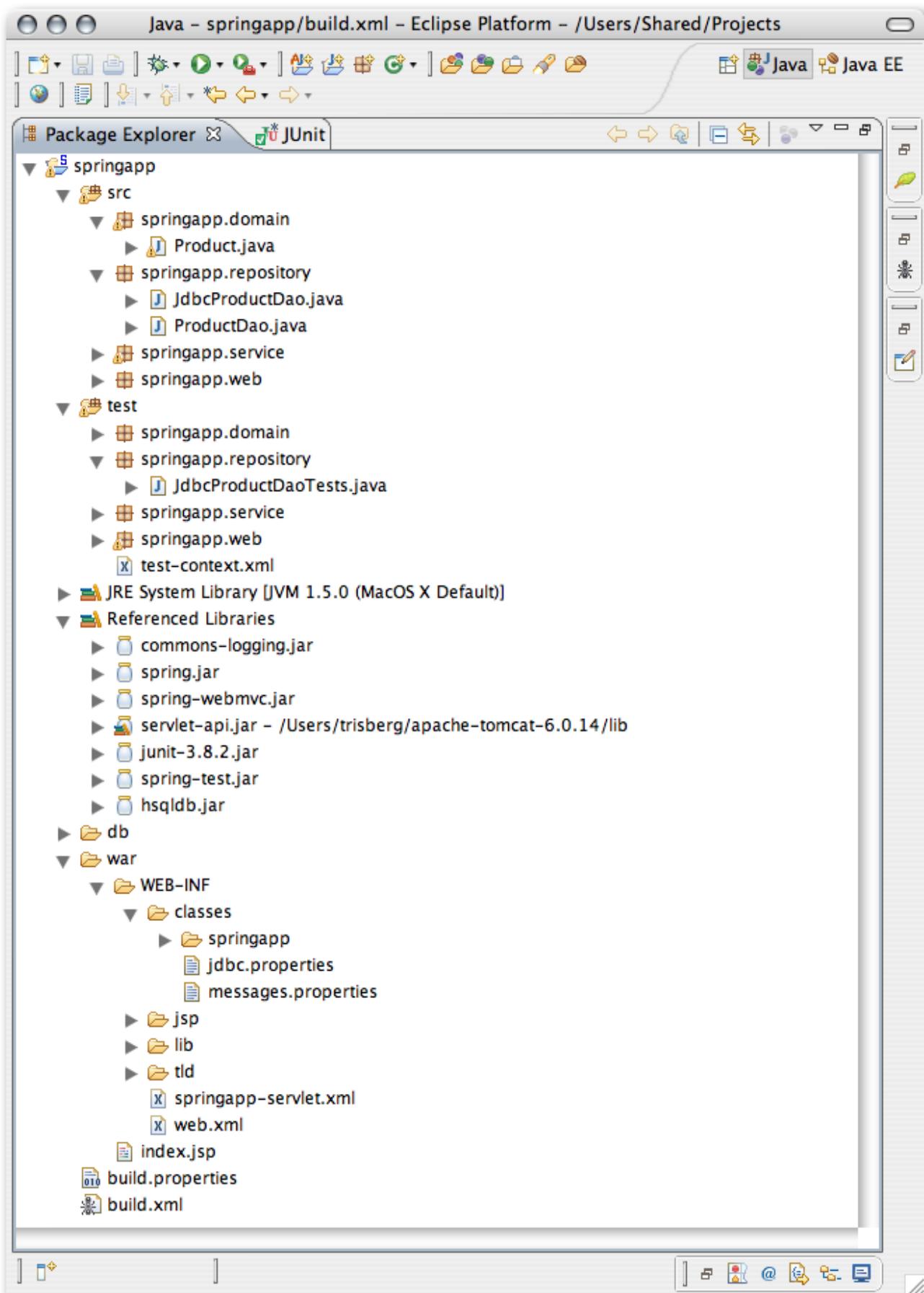
Chapter 6. Integrating the Web Application with the Persistence Layer

This is Part 6 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. In Part 5 we developed the persistence layer. It is now time to integrate all this into a complete web application.

6.1. Modify service layer

If we structured our application properly, we should only have to change the service layer classes to take advantage of the database persistence. The view and controller classes should not have to be modified, since they should be unaware of any implementation details of the service layer. So let's add the persistence to the ProductManager implementation. We modify the `SimpleProductManager` and add a reference to a `ProductDao` interface plus a setter method for this reference. Which implementation we actually use here should be irrelevant to the ProductManager class, and we will set this through a configuration option. We also change the `setProducts` method to a `setProductDao` method so we can inject an instance of the DAO class. The `getProducts` method will now use the DAO to retrieve a list of products. Finally, the `increasePrices` method will now get the list of products and then after the price have been increased the product will be stored in the database using the `saveProduct` method on the DAO.

`'springapp/src/springapp/service/SimpleProductManager.java'`:

```java
package springapp.service;

import java.util.List;

import springapp.domain.Product;
import springapp.repository.ProductDao;

public class SimpleProductManager implements ProductManager {

    // private List<Product> products;
    private ProductDao productDao;

    public List<Product> getProducts() {
        // return products;
        return productDao.getProductList();
    }

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

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

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

}
```

6.2. Fix the failing tests

We rewrote the `SimpleProductManager` and now the tests will of course fail. We need to provide the `ProductManager` with an in-memory implementation of the `ProductDao`. We don't really want to use the real DAO here since we'd like to avoid having to access a database for our unit tests. We will add an internal class called `InMemoryProductDao` that will hold on to a list pf products provided in the constructor. This in-memory class has to be passed in when we create a new `SimpleProductManager`.

`'springapp/test/springapp/repository/InMemoryProductDao.java'`:

```
package springapp.repository;

import java.util.List;

import springapp.domain.Product;

public class InMemoryProductDao implements ProductDao {

    private List<Product> productList;

    public InMemoryProductDao(List<Product> productList) {
        this.productList = productList;
    }

    public List<Product> getProductList() {
        return productList;
    }

    public void saveProduct(Product prod) {
    }

}
```

And here is the modified `SimpleProductManagerTests`:

`'springapp/test/springapp/service/SimpleProductManagerTests.java'`:

```
package springapp.service;

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

import springapp.domain.Product;
import springapp.repository.InMemoryProductDao;
import springapp.repository.ProductDao;

import junit.framework.TestCase;

public class SimpleProductManagerTests 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();
```

```java
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

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

        ProductDao productDao = new InMemoryProductDao(products);
        productManager.setProductDao(productDao);
        //productManager.setProducts(products);
    }

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

    public void testIncreasePriceWithEmptyListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.setProductDao(new InMemoryProductDao(new ArrayList<Product>()));
            //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 also need to modify the `InventoryControllerTests` since that class also uses the `SimpleProductManager`. Here is the modified `InventoryControllerTests`:

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

```
package springapp.web;

import java.util.ArrayList;
import java.util.Map;

import org.springframework.web.servlet.ModelAndView;

import springapp.domain.Product;
import springapp.repository.InMemoryProductDao;
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();
        SimpleProductManager spm = new SimpleProductManager();
        spm.setProductDao(new InMemoryProductDao(new ArrayList<Product>()));
        controller.setProductManager(spm);
        //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);
    }
}
```

6.3. Create new application context for service layer configuration

We saw earlier that it was fairly easy to modify the service layer to use the database persistence. This was because it is decoupled from the web layer. It's now time to decouple or configuration of the service layer from the web layer as well. We will remove the productManager configuration and the list of products from the springapp-servlet.xml configuration file. This is what this file looks like now:

`'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="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>
```

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

</beans>
```

We still need to configure the service layer and we will do that in its own application context file. This file is called `'applicationContext.xml'` and it will be loaded via a servlet listener that we will define in `'web.xml'`. All bean configured in this new application context will be available to reference from any servlet context.

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

```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" >

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <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>
```

Now we create a new `'applicationContext.xml'` file in the `'war/WEB-INF'`" directory.

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

```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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <!-- the parent application context definition for the springapp application -->
```

```
    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="productDao" ref="productDao"/>
    </bean>

    <bean id="productDao" class="springapp.repository.JdbcProductDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

6.4. Add transaction and connection pool configuration to application context

Any time you persist data in a database its best to use transactions to ensure that all your updates are perform or none are completed. You want to avoid having half your updates persisted while the other half failed. Spring provides an extensive range of options for how to provide transaction management. The reference manual covers this in depth. Here we will make use of one way of providing this using AOP (Aspect Oriented Programming) in the form of a transaction advice and an ApectJ pointcut to define where the transactions should be applied. If you are interested in how this works in more depth, take a look at the reference manual. We are using the new namespace support introduced in Spring 2.0. The "aop" and "tx" namespaces make the configuration entries much more concise compared to the traditional way using regular "<bean>" entries.

```
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <aop:config>
        <aop:advisor pointcut="execution(* *..ProductManager.*(..))" advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="save*"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>
```

The pointcut applies to any method called on the ProductManager interface. The advice is a transaction advice that applies to methods with a name starting with 'save'. The default transaction attributes of REQUIRED applies since no other attribute was specified. The advice also applies "read-only" transactions on any other methods that are adviced via the pointcut.

We also need to define a connection pool. We are using the DBCP connection pool from the Apache Jakarta project. We are reusing the `jdbc.properties` file we created in Part 5.

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

    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>
```

For all this to work we need some additional jar files copied to the `'WEB-INF/lib'` directory. Copy `aspectjweaver.jar` from the `'spring-framework-2.5/lib/aspectj'` directory and `commons-dbcp.jar` and `commons-pool.jar` from the `'spring-framework-2.5/lib/jakarta-commons'` directory to the `'springapp/war/WEB-INF/lib'` directory.

Here is the final version of our `'applicationContext.xml'` file:

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

```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"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
          http://www.springframework.org/schema/tx
          http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

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

    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="productDao" ref="productDao"/>
    </bean>

    <bean id="productDao" class="springapp.repository.JdbcProductDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

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

    <bean id="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <aop:config>
        <aop:advisor pointcut="execution(* *..ProductManager.*(..))" advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="save*"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>

</beans>
```

6.5. Final test of the complete application

Now it's finally time to see if all of these pieces will work together. Build and deploy your finished application

and remember to have the database up and running. This is what you should see when pointing the web browser at the application after it has reloaded:
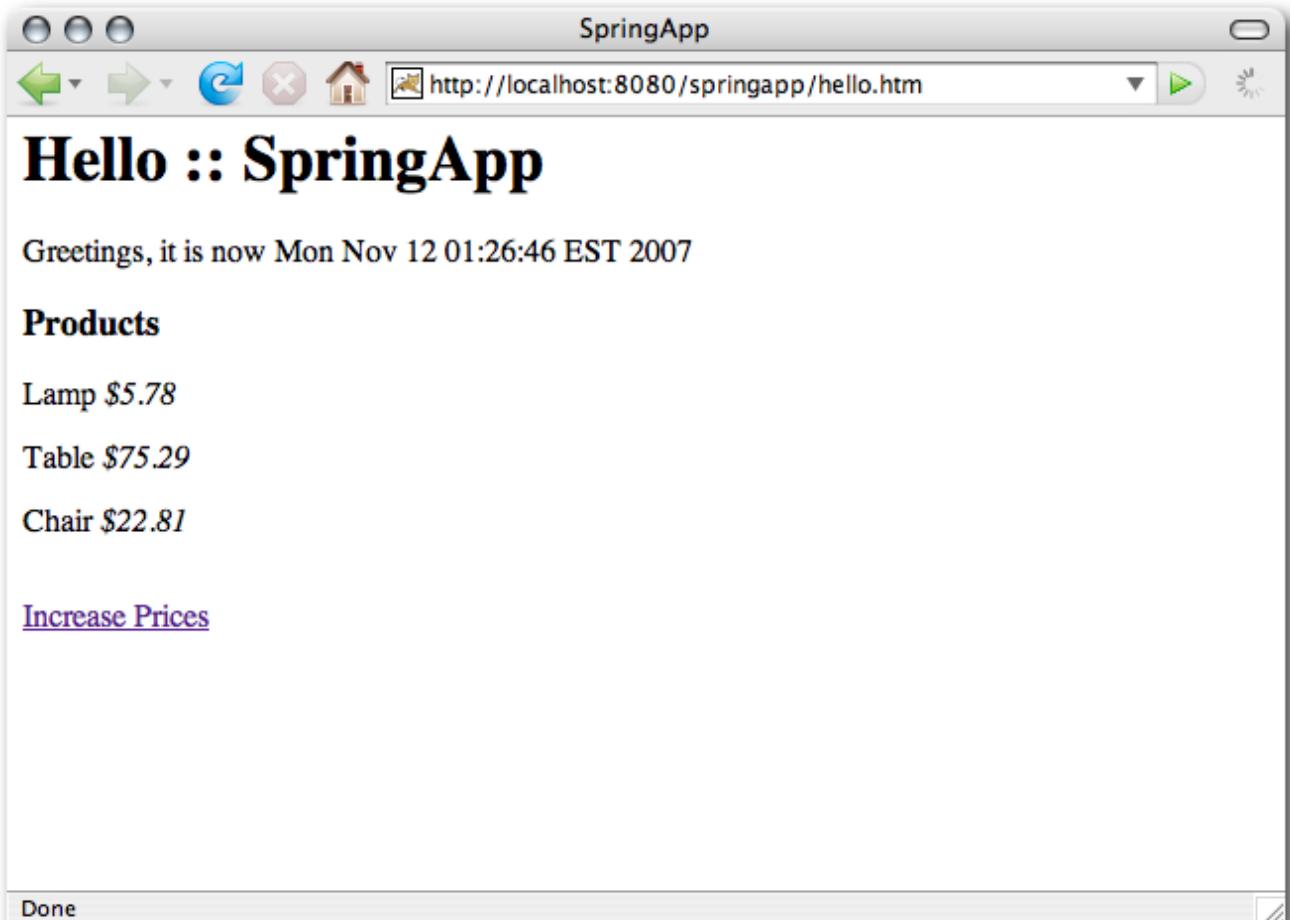
The completed application

Looks just the same as it did before. We did add persistence though, so if you shut down the application your price increases will not be lost. They are still there when you start the application back up.

A lot of work for a very simple application, but it was never our goal to just write this application. The goal was to show how to go about creating a Spring MVC application from scratch and we know that the applications you will create are much more complex. The same steps apply though and we hope you have gained enough knowledge to make it easier getting started to use Spring.

6.6. Summary

We have completed all three layers of the application -- the web layer, the service layer and the persistence layer. In this last part we reconfigured the application.

1. First we modified the service layer to use the ProductDao.

2. We then had to fix some failing service and web layer tests.

3. Next we introduced a new applicationContext to separate the service and persistence layer configuration from the web layer configuration.

4. We also defined some transaction management for the service layer and configured a connection pool for

the database connections.

5.   Finally we built the reconfigured application and tested that it still worked.

Below is a screen shot of what your project directory structure should look like after following the above instructions.
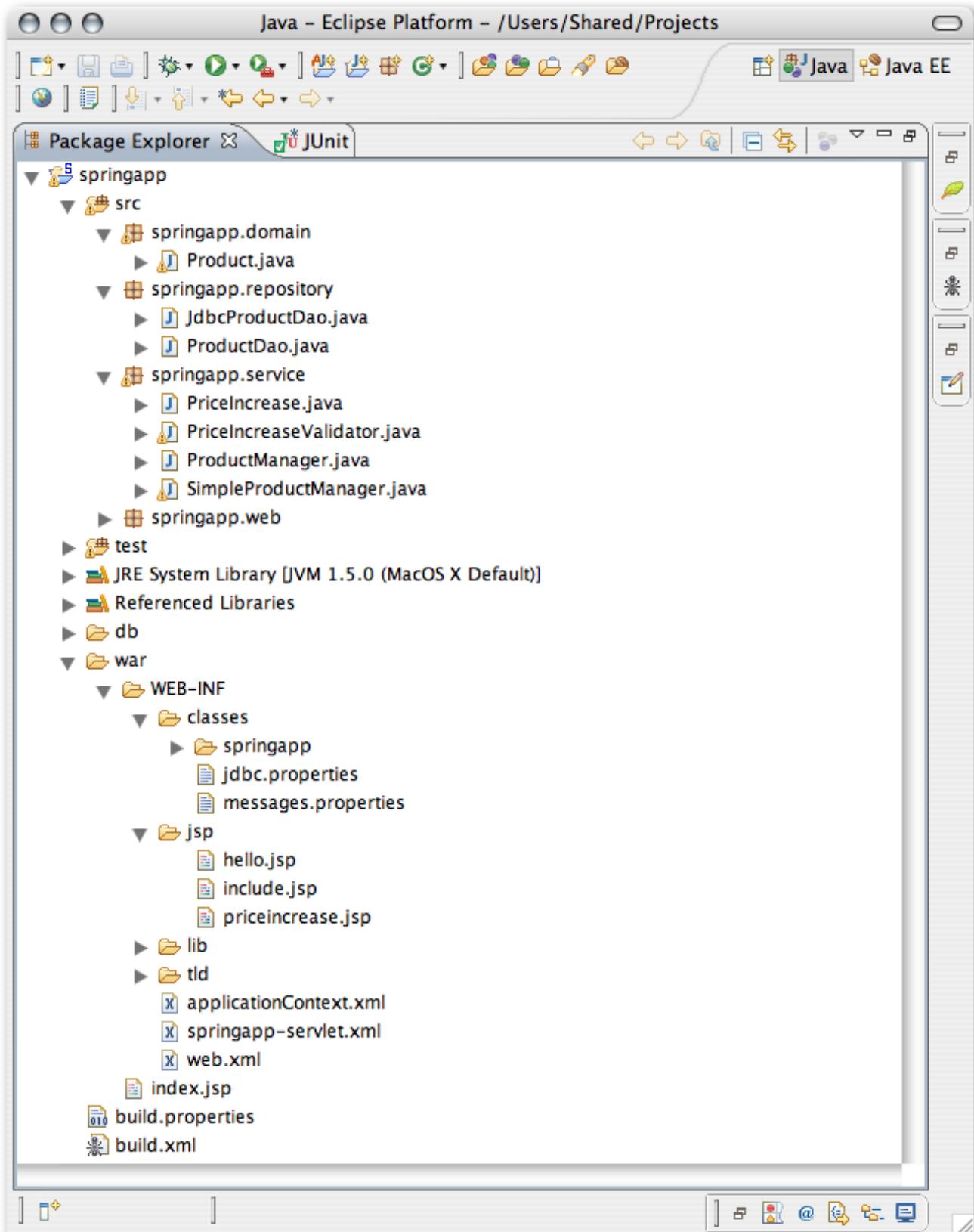
Appendix A. Build Scripts

Complete `build.xml` listing:

```xml
<?xml version="1.0"?>

<project name="springapp" basedir="." default="usage">
    <property file="build.properties"/>

    <property name="src.dir" value="src"/>
    <property name="web.dir" value="war"/>
    <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
    <property name="name" value="springapp"/>

    <path id="master-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <!-- We need the servlet API classes: -->
        <!--  * for Tomcat 5/6 use servlet-api.jar -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="servlet*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
        <pathelement path="${test.dir}"/>
    </path>

    <target name="usage">
        <echo message=""/>
        <echo message="${name} build file"/>
        <echo message="-----------------------------------"/>
        <echo message=""/>
        <echo message="Available targets are:"/>
        <echo message=""/>
        <echo message="build     --> Build the application"/>
        <echo message="deploy    --> Deploy application as directory"/>
        <echo message="deploywar --> Deploy application as a WAR file"/>
        <echo message="install   --> Install application in Tomcat"/>
        <echo message="reload    --> Reload application in Tomcat"/>
        <echo message="start     --> Start Tomcat application"/>
        <echo message="stop      --> Stop Tomcat application"/>
        <echo message="list      --> List Tomcat applications"/>
        <echo message=""/>
    </target>

    <target name="build" description="Compile main source tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
               deprecation="false" optimize="false" failonerror="true">
            <src path="${src.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <target name="deploy" depends="build" description="Deploy application">
        <copy todir="${deploy.path}/${name}" preservelastmodified="true">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </copy>
    </target>

    <target name="deploywar" depends="build" description="Deploy application as a WAR file">
        <war destfile="${name}.war"
             webxml="${web.dir}/WEB-INF/web.xml">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </war>
        <copy todir="${deploy.path}" preservelastmodified="true">
            <fileset dir=".">
                <include name="*.war"/>
            </fileset>
```

```
        </copy>
    </target>

    <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>

    <property name="test.dir" value="test"/>

    <target name="buildtests" description="Compile test tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
            deprecation="false" optimize="false" failonerror="true">
            <src path="${test.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <path id="test-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
        <pathelement path="${test.dir}"/>
        <pathelement path="${web.dir}/WEB-INF/classes"/>
    </path>

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/*Tests.*"/>
                    <exclude name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ********************************************************
            ********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ********************************************************
            ********************************************************
        </fail>
    </target>

    <target name="dbTests" depends="build, buildtests,dropTables,createTables,loadData"
            description="Run db tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>
```

```
                <batchtest>
                    <fileset dir="${build.dir}">
                        <include name="**/Jdbc*Tests.*"/>
                    </fileset>
                </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ************************************************************
            ************************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ************************************************************
            ************************************************************
        </fail>
    </target>

    <target name="createTables">
        <echo message="CREATE TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/create_products.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="dropTables">
        <echo message="DROP TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

    DROP TABLE products;

        </sql>
    </target>

    <target name="loadData">
        <echo message="LOAD DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/load_data.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="printData">
        <echo message="PRINT DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             print="true">
            <classpath refid="master-classpath"/>

    SELECT * FROM products;

        </sql>
    </target>

    <target name="clearData">
        <echo message="CLEAR DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
```

```
         onerror="continue">
            <classpath refid="master-classpath"/>

        DELETE FROM products;

        </sql>
    </target>

    <target name="shutdownDb">
        <echo message="SHUT DOWN DATABASE USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue">
            <classpath refid="master-classpath"/>

        SHUTDOWN;

        </sql>
    </target>

<!-- ============================================================ -->
<!-- Tomcat tasks - remove these if you don't have Tomcat installed -->
<!-- ============================================================ -->

    <path id="catalina-ant-classpath">
        <!-- We need the Catalina jars for Tomcat -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="catalina-ant.jar"/>
        </fileset>
    </path>

    <taskdef name="install" classname="org.apache.catalina.ant.InstallTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="start" classname="org.apache.catalina.ant.StartTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="stop" classname="org.apache.catalina.ant.StopTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>

    <target name="install" description="Install application in Tomcat">
        <install url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"
                war="${name}"/>
    </target>

    <target name="reload" description="Reload application in Tomcat">
        <reload url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
    </target>

    <target name="start" description="Start Tomcat application">
        <start url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
    </target>

    <target name="stop" description="Stop Tomcat application">
        <stop url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"
                path="/${name}"/>
```

```
    </target>

    <target name="list" description="List Tomcat applications">
        <list url="${tomcat.manager.url}"
                username="${tomcat.manager.username}"
                password="${tomcat.manager.password}"/>
    </target>

<!-- End Tomcat tasks -->

</project>
```

Complete `build.properties` listing:

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret

db.driver=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:hsql://localhost
db.user=sa
db.pw=
```