Desarrollando una aplicacion Spring Framework MVC paso a paso

Autores

Thomas Risberg, Rick Evans, Portia Tung

Traducción

David Marco Palao (programacion@davidmarco.es)

Versión de página única y versión PDF

David Villacé Hernández (vilherda@gmail.com)

2.5

Tabla de Contenidos

Descripcion

Este documento es una guia paso a paso sobre como desarrollar una aplicacion web, partiendo de cero, usando Spring Framework.

Se asume que posees un conocimiento superficial de Spring, por lo que este tutorial te sera util si estas aprendiendo o investigando Spring. Durante el tiempo que trabajes a traves del material presentado en el tutorial, podras ver como encajan diversas partes de Spring Framework dentro de una aplicacion web Spring MVC, como Inversion de Control (Inversion of Control - IoC), Programacion Orientada a Aspectos (Aspect-Oriented Programming - AOP), asi como las diversas librerias de servicios (como la libreria JDBC).

Spring provee diversas opciones para configurar tu aplicacion. La forma mas popular es usando archivos XML. Esta es la forma mas tradicional, soportada desde la primera version de Spring. Con la introduccion de Anotaciones en Java 5, ahora disponemos de una manera alternativa de configurar nuestras aplicaciones Spring. La nueva version Spring 2.5 introduce un amplio soporte para configurar una aplicacion web mediante anotaciones.

Este documento usa el estilo tradicional de configuracion, mediante XML. Estamos trabajando en una *"Edicion con Anotaciones"* de este documento, y esperamos publicarla en un futuro cercano.

Ten en cuenta que no se tratara ninguna informacion en profundidad en este tutorial, asi como ningun tipo de teoria; hay multitud de libros dispobibles que cubren Spring en profundidad; siempre que una nueva clase o caracteristica sea usada en el tutorial, se mostraran enlaces a la seccion de documentacion de Spring, donde la clase o caracteristica es tratada en profundidad.

1. Contenido

La siguiente lista detalla todas las partes de Spring Framework que son cubiertas a lo largo del tutorial.

- Inversion de Control (IoC)
- El framework Spring Web MVC
- Acceso a Datos mediante JDBC
- Integracion mediante tests
- Manejo de transacciones

2. Software Requerido

Se requiere el siguiente software y su adecuada configuracion en el entorno. Deberias sentirte razonablemente confortable usando las siguiente tecnologias.

- Java SDK 1.5
- Ant 1.7
- Apache Tomcat 6.0.14
- Eclipse 3.3 (Recomendado, pero no necesario)

Eclipse 3.3 Europa (http://www.eclipse.org/europa) junto con el proyecto Web Tools Platform (WTP) (http://www.eclipse.org/webtools) y el proyecto Spring IDE (http://www.springide.org) proporcionan un excelente entorno para el desarrollo web. Por supuesto, puedes usar cualquier variacion de las versiones de software anteriores. Si quieres usar NetBeans o IntelliJ en lugar de Eclipse, o Jetty en lugar de Tomcat, ciertos pasos de este tutorial no se corresponderan directamente con tu entorno, pero deberias ser capaz de seguir adelante de todas maneras.

3. La aplicacion que vamos a construir

La aplicacion que vamos a construir desde cero a lo largo de este tutorial es un sistema de mantenimiento de inventario *muy basico*. Este sistema de mantenimiento de

inventario esta muy limitado en alcance y funcionalidad; debajo puedes ver un diagrama de casos de uso ilustrando los sencillos casos de uso que implementaremos. La razon por la que la aplicacion es tan limitada es que puedas concentrarte en las caracteristicas especificas de Spring Web MCV y Spring, y olvidar los detalles mas sutiles del mantenimiento de inventario.

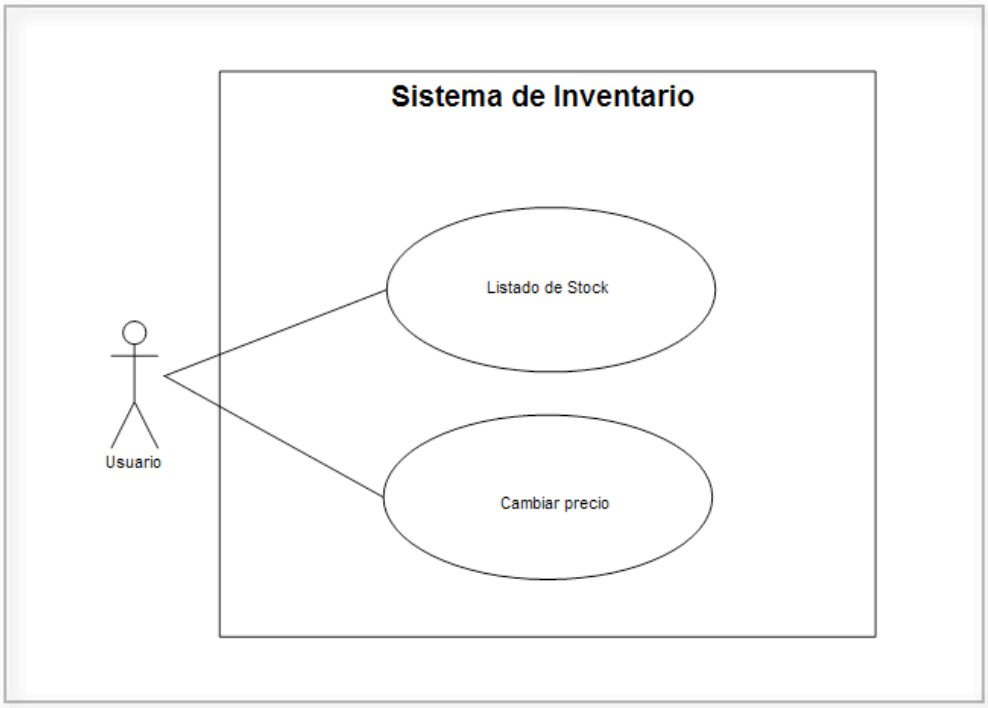


Diagrama de casos de uso de un sistema de mantenimiento de inventario

Comenzaremos configurando la estructura basica de los directorios para nuestra aplicacion, descargando las librerias necesarias, configurando nuestros scripts Ant, etc. El primer paso nos proporcionara una base solida sobre la que desarrollar de forma adecuada las secciones 2, 3, y 4.

Una vez terminada la configuracion basica, introduciremos Spring, comenzando con el framework Spring Web MVC. Usaremos Spring Web MVC para mostrar el stock inventariado, el cual implicara escribir algunas clases simples en Java y algunos JSP. Entonces introduciremos acceso de datos y persistencia en nuestra aplicacion, usando para ello el soporte que ofrece Spring para JDBC.

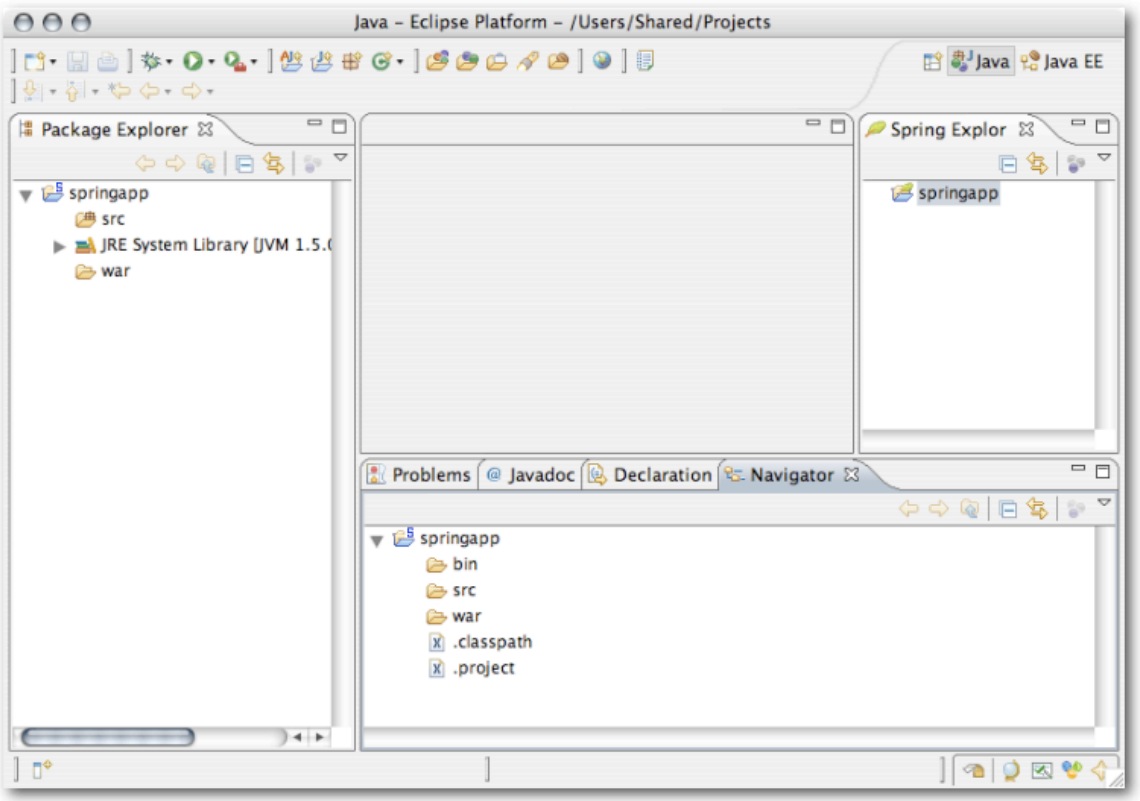
Una vez hayamos finalizado todos los pasos del tutorial, tendremos una aplicacion que realiza un mantenimiento basico de stock, incluyendo listados de stock y permitiendo el incremento de precios de dicho stock.

Capitulo 1. Aplicacion Base y Configuracion del Entorno

1.1. Crear la estructura de directorios del proyecto

Necesitamos crear un lugar donde alojar todos los archivos que vayamos creando, asi que comenzaremos creando un directorio llamado `'springapp'`. La decision sobre donde crear este directorio esta en tu mano; nosotros hemos creado el nuestro dentro del directorio `'Projects'` que anteriormente habiamos creado en `'home'`, por lo que la ruta completa a nuestro directorio del proyecto es `'$HOME/Projects/springapp'`. Dentro de este directorio, vamos a crear un subdirectorio llamado`'src'` donde guardar todos los archivos de codigo fuente Java. De nuevo en el directorio `'$HOME/Projects/springapp'` creamos otro subdirectorio llamado `'war'`. Este directorio almacenara todo lo que debe incluir el archivo WAR que usaremos para almacenar y desplegar rapidamente nuestra aplicacion. Todos los archivos que no sean codigo fuente Java, como archivos JSP y de configuracion, se alojaran en el directorio `'war'` .

Debajo puedes ver una captura de pantalla mostrando como quedaria la estructura de directorios si has seguido las instrucciones correctamente. *(La imagen muestra dicha estructura desde el IDE Eclipse: no se necesita usar Eclipse para completar este tutorial, pero usandolo podras hacerlo de manera mucho mas sencilla).*



La estructura de directorios del proyecto

1.2. Crear `'index.jsp'`

Puesto que estamos creando una aplicacion web, comencemos creando un archivo JSP muy simple llamado `'index.jsp'` en el directorio `'war'`. El archivo `'index.jsp'` es el punto de entrada a nuestra aplicacion.

`'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>
```

Para tener una aplicacion web completa vamos a crear un directorio llamado **'WEB-INF'** dentro del directorio **'war'**, y dentro de este nuevo directorio creamos un archivo llamado **'web.xml'**.

'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. Desplegar la aplicacion en Tomcat

Escribamos ahora un script para Ant que utilizaremos a lo largo de todo este tutorial. Este script para Ant contendra la informacion necesaria para compilar, construir y desplegar la aplicacion automaticamente. Ademas, otro script para Ant sera creado y usado para tareas especificas del servidor.

'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>
```

Si estas usando un servidor de aplicaciones web distinto, puedes eliminar las tareas especificas de Tomcat que hay al final del script. Tendras que confiar en que tu servidor soporte despliegue en caliente, o de lo contrario tendras que parar e iniciar tu aplicacion manualmente.

Si estas usando un IDE, es posible que encuentre errores en el archivo `'build.xml'`, como los objetivos de Tomcat. Ignoralos, el archivo listado arriba es correcto.

El archivo de script para Ant listado arriba contiene toda la configuracion de objetivos que vamos a necesitar para hacer nuestro esfuerzo de desarrollo mas facil. No vamos a explicar este script en detalle, puesto que la mayor parte de el es sobre todo configuracion estandard para Ant y Tomcat. Selecccciona todo el script, copialo, y pegalo en un nuevo archivo de texto llamado `'build.xml'` que debes guardar en tu directorio principal de desarrollo. Tambien necesitamos un archivo `'build.properties'` que modificaremos para que coincida con nuestra instalacion del servidor. Este archivo permanecera en el mismo directorio donde hemos guardado el archivo `'build.xml'`.

`'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
```

*Si estas en un equipo donde no eres el usuario propietario de la instalacion de Tomcat, entonces dicho usuario debe garantizarte el acceso sin restriccciones al directorio `'webapps'`, o el propietario debe crear un nuevo directorio llamado `'springapp'` en el directorio `'webapps'` de Tomcat, y crear los permisos necesarios para que puedas desplegar la aplicacion en este directorio. En Linux, escribe el comando **'chmod a+rwx springapp'** para dar todos los permisos a este directorio.*

Para crear un usuario de Tomcat llamado 'tomcat' con contraseña 's3cret', edita el archivo de usuarios de Tomcat `'appserver.home/conf/tomcat-users.xml'` y añade una nueva entrada de usuario.

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

Ahora ejecutaremos Ant para estar seguros que todo funciona bien. Debes estar situado en el directorio `'springapp'`.

Abre una consola de shell (o prompt) y ejecuta el comando **'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
```

La ultima cosa que debemos hacer es construir y desplegar la aplicacion. Para ello, simplemente ejecuta Ant y especifica `'deploy'` o `'deploywar'` como objetivo.

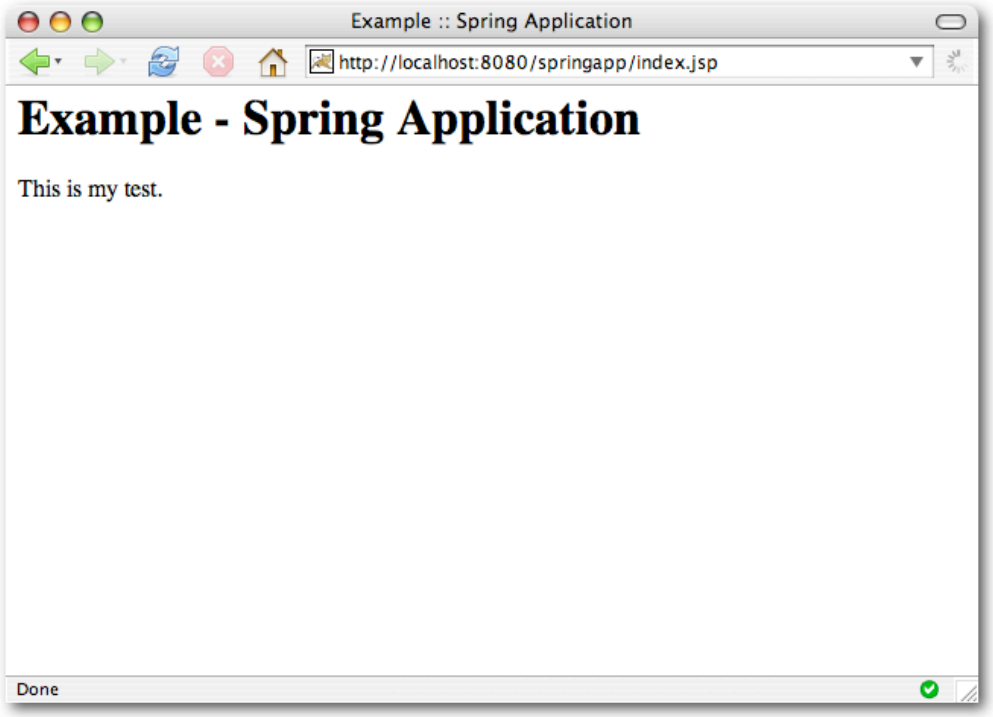
```
$ ant deploy
Buildfile: build.xml

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. Comprobar que la aplicacion funciona

Inicia tomcat ejecutando el archivo `'${appserver.home}/bin/startup.bat'`. Para asegurarnos que podemos acceder a la aplicacion, ejecuta la tarea `'list'` desde nuestro archivo de contruccion Ant para ver si Tomcat puede encontrar la aplicacion.

```
$ 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
```

Ahora puedes abrir un navegador y acceder a la pagina de inicio de la aplicacion en la siguiente URL: http://localhost:8080/springapp/index.jsp.



La Pagina de inicio de la aplicacion

1.5. Descargar Spring Framework

Si aun no has descargado Spring Framework, ahora es el momento de hacerlo. Nosotros usamos actualmente la version 2.5 de Spring Framework, que puede ser descargada desde http://www.springframework.org/download. Descomprime el archivo que has descargado en una carpeta, ya que mas tarde vamos a usar varios archivos que estan en su interior.

Esto completa la configuracion de entorno necesaria, asi que ya podemos comenzar a desarrollar nuestra aplicacion Spring Framework MVC.

1.6. Modificar `'web.xml'` en el directorio `'WEB-INF'`

Situate en el directorio **`'springapp/war/WEB-INF'`**. Modifica el archivo **`'web.xml'`** que hemos creado anteriormente. Vamos a definir un `DispatcherServlet` (tambien llamado `'Controlador Frontal'` (Crupi et al)). Su mision sera controlar hacia donde seran enrutadas todas nuestras solicitudes basandose en informacion que introduciremos posteriormente. La definicion del servlet tendra como acompañante una entrada `<servlet-mapping/>` que mapeara las URL que queremos que apunten a nuestro servlet. Hemos decidido permitir que cualquier URL con una extension de tipo `'.htm'` sea enrutada hacia el servlet `'springapp'` (`DispatcherServlet`).

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

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

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

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

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

  <welcome-file-list>
```

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

</web-app>
```

A continuacion, crea un archivo llamado **'springapp-servlet.xml'** en el directorio **'springapp/war/WEB-INF'**. Este archivo contiene las definiciones de beans (POJO's) usados por `DispatcherServlet`. Este archivo es el `WebApplicationContext` donde situaremos todos los componentes web. El nombre de este archivo esta determinado por el valor del elemento `<servlet-name/>` en **'web.xml'**, con la palabra `'-servlet'` agregada al final (por lo tanto **'springapp-servlet.xml'**). Esta es la convencion estardar para nombrar archivos del framework Spring MVC. Ahora añade una definicion de bean llamada `'/hello.htm'` y especifica su clase como `springapp.web.HelloController`. Esto define el controlador que nuestra aplicacion usara para dar servicio a solicitudes con el correspondiente mapeo de URL de `'/hello.htm'`. El framework Spring MVC usa una clase que implementa la interface `HandlerMapping` para definir el mapeo entre la URL solicitada y el objeto que va a manejar la solicitud (manejador). Al contrario que `DispatcherServlet`, `HelloController` es el encargado de manejar las solicitudes para una pagina concreta del sitio web, y es conocido como el `'Controlador de Pagina'` (Fowler). El `HandlerMapping` por defecto que `DispatcherServlet` usa es `BeanNameUrlHandlerMapping`; esta clase usa el nombre del bean para mapear la URL de la solicitud, por lo que `DispatcherServlet` conocera que controlador debe ser invocado para manejar diferentes URLs.

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

</beans>
```

1.7. Copiar librerias a **'WEB-INF/lib'**

Primero crea un directorio llamado **'lib'** dentro del directorio **'war/WEB-INF'** . Ahora, desde la distribucion de Spring, copia los archivos **spring.jar** (desde **spring-framework-2.5/dist**) y **spring-webmvc.jar** (desde **spring-framework-2.5/dist/modules**) al recien creado directorio **'war/WEB-INF/lib'**. Ademas, copia el archivo **commons-logging.jar** (desde **spring-framework-2.5/lib/jakarta-commons**) tambien al directorio **'war/WEB-INF/lib'**. Estos archivos jar seran desplegados y usados en el servidor durante el proceso de construccion.

1.8. Crear el `Controlador`

Vamos a crear una instancia de la clase `Controller` – a la que llamaremos `HelloController`, y que estara definida dentro del paquete `'springapp.web'`. Primero, crea los directorios necesarios para que coincidan con el arbol del paquete. A continuacion, crea el archivo **'HelloController.java'** y situalo en el recien citado directorio **'src/springapp/web'** .

'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;

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");
    }

}
```

Esta implementacion del controlador `Controller` es muy basica. Mas tarde la iremos expandiendo, asi como extendiendo parte de la implementacion base provista por Spring. En Spring Web MVC, `Controller` *maneja* las solicitudes y devuelve un objeto `ModelAndView` - en este caso, uno llamado **`'hello.jsp'`** el cual es ademas el nombre del archivo JSP que vamos a crear a continuacion. El modelo que esta clase devuelve es resuelto via `ViewResolver`. Puesto que no hemos definido explicitamente un `ViewResolver`, vamos a obtener uno por defecto de Spring que simplemente redigira a una direccion URL que coincida con el nombre de la vista especificada. Mas tarde modificaremos esto. Ademas, hemos especificado un logger de manera que podemos verificar que pasamos por el manejador en cada momento. Usando Tomcat, estos mensajes de log deben mostrarse en el arhivo de log **`'catalina.out'`** que puede ser encontrado en el directorio **`'${appserver.home}/log'`** de tu instalacion de Tomcat.

*Si estas usando un IDE, configura las dependencias del proyecto añadiendo los archivos jar desde el directorio **`'lib'`**. Añade ademas el archivo **`servlet-api.jar`** desde el directorio **`'lib'`** de tu contenedor de servlets (`'${appserver.lib}'`). Añadiendo estos archivos a las dependencias del proyecto, deberian funcionar todas las sentencias import del archivo **`'HelloController.java'`**.*

1.9. Escribir un test para el `Controlador`

Los tests son una parte vital del desarrollo de software. Es ademas una de las practicas fundamentales en Desarrollo Agil. El mejor momento para escribir tests es durante el desarrollo, no despues, de manera que aunque nuestro controlador no contiene logica demasiado compleja vamos a escribir un test para probarlo. Esto nos permitira hacer cambios en el futuro con total seguridad. Vamos a crear un nuevo directorio bajo **`'springapp'`** llamado **`'test'`**. Aqui es donde alojaremos todos nuestros tests, en una estructura de paquetes que sera identica a la estructura de paquetes que tenemos en **`'springapp/src'`**.

Crea una clase de test llamada **`'HelloControllerTests'`** y haz que extienda la clase de JUnit `TestCase`. Esta es una unidad de test que verifica que el nombre de vista devuelto por `handleRequest()` coincide con el nombre de la vista que esperamos: **`'hello.jsp'`**.

`'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.jsp", modelAndView.getViewName());
    }
}
```

Para ejecutar el test (y todos los tests que escribamos en el futuro), necesitamos añadir una tarea de test a nuestro script de Ant **`'build.xml'`**. Primero, copiamos **`junit-3.8.2.jar`** desde **`'spring-framework-2.5/lib/junit'`** al directorio **`'war/WEB-INF/lib'`**. En lugar de crear una tarea unica para compilar los tests y a continuacion ejecutarlos, vamos a separar el proceso en dos tareas diferentes: `'buildtests'` y `'tests'`, el cual depende de `'buildtests'`.

*Si estas usando un IDE, tal vez prefieras ejecutar los tests desde el IDE. Configura las dependencias de tu proyecto añadiendo **`junit-3.8.2.jar`**.*

`'springapp/build.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>
```

Ahora ejecuta la tarea de Ant `'tests'` y el test debe pasar.

```
$ 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
```

Otra de las mejores practicas dentro del Desarrollo Agil es *Integracion Continua*. Es una muy buena idea asegurar que tus tests se ejecutan con cada construccion (idealmente como construccion automatica del proyecto) de manera que sepas que la logica de tu aplicacion se comporta como se espera de ella a traves de la evolucion del codigo.

1.10. Crear la `Vista`

Ahora es el momento de crear nuestra primera vista. Como hemos mencionado antes, estamos redirigiendo hacia una pagina JSP llamada **`'hello.jsp'`**. Para empezar, crearemos este fichero en el directorio **`'war'`**.

`'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. Compilar y desplegar la aplicacion

Ejecuta la tarea `'deploy'` en Ant (la cual invoca la tarea `'build'`), y a continuacion `'reload'` del archivo **`'build.xml'`** . Esto forzara la construccion y recarga de la aplicacion en Tomcat. Tenemos que comprobar la salida de Ant y los logs de Tomcat para verificar cualquier posible error de despliegue – como errores tipograficos en los archivos de arriba y clases o archivos jar no encontrados.

Aqui tienes un ejemplo de salida del script de construccion Ant:

```
$ 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
```

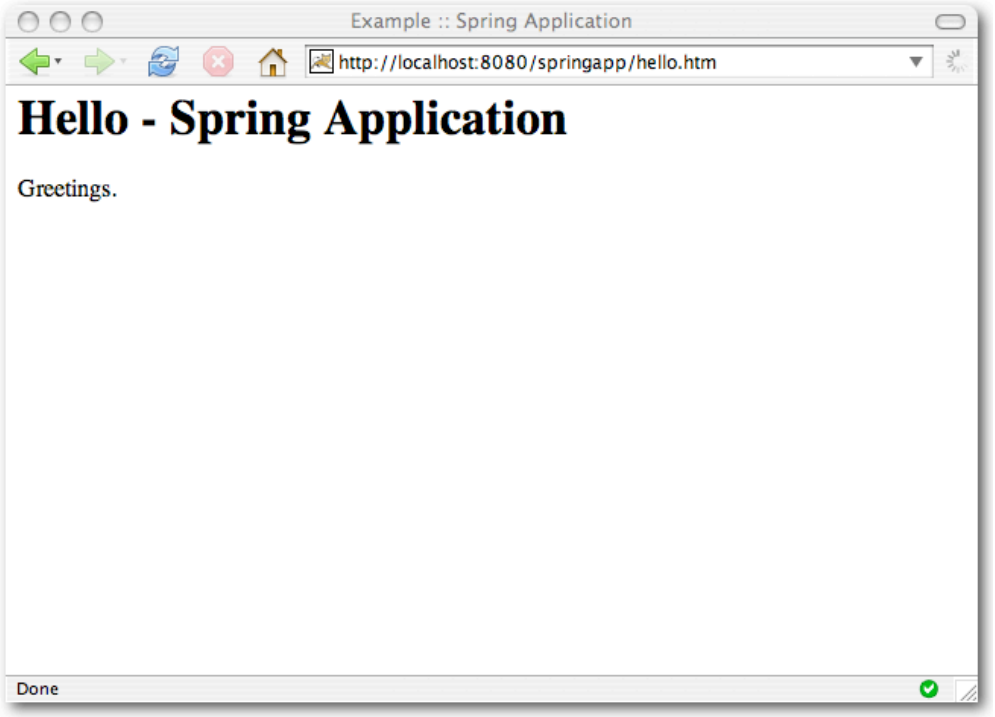
Y aqui un extracto del archivo de log **`'catalina.out'`**.

```
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. Probar la aplicacion

Probemos esta nueva version de la aplicacion.

Abre un navegador y navega hasta http://localhost:8080/springapp/hello.htm.



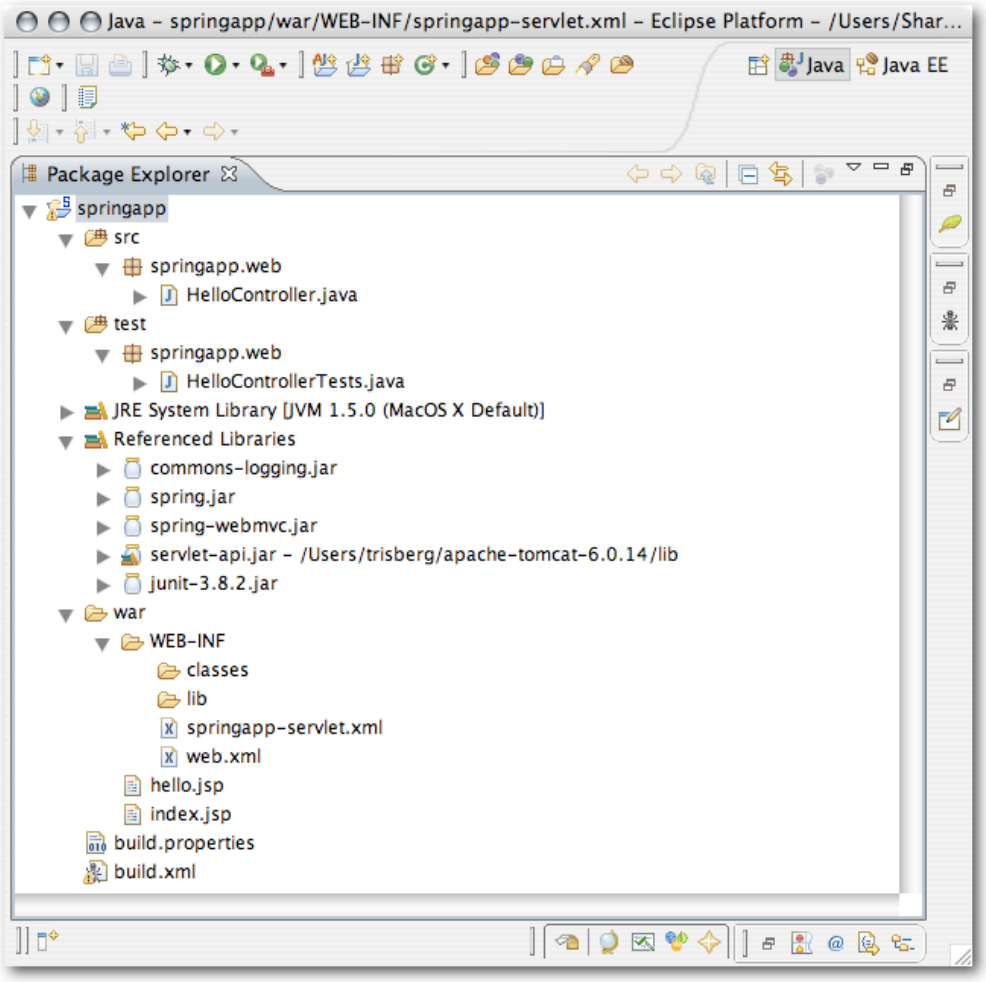
La aplicacion actualizada

1.13. Resumen

Echemos un vistazo rapido a las partes de nuestra aplicacion que hemos creado hasta ahora.

- Una pagina de inicio, `'index.jsp'`, la pagina de bienvenida de nuestra aplicacion. Fue usada para comprobar que nuestra configuracion era correcta. Mas tarde la cambiaremos para proveer un enlance a nuestra aplicacion.
- Un controlador frontal, `DispatcherServlet` con el correspondiente archivo de configuracion `'springapp-servlet.xml'`.
- Un controlador de pagina, `HelloController`, con funcionalidad limitada – simplemente devuelve un objeto `ModelAndView`. Actualmente tenemos un modelo vacio, mas tarde proveeremos un modelo completo.
- Una unidad de test para la pagina del controlador, `HelloControllerTests`, para verificar que el nombre de la vista es el que esperamos.
- Una vista, `'hello.jsp'`, que de nuevo es extremadamente sencilla. Las buenas noticias son que el conjunto de la aplicacion funciona y que estamos listos para añadir mas funcionalidad.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 1

Capitulo 2. Desarrollando y Configurando la Vista y el Controlador

Esta es la Parte 2 del tutorial paso a paso sobre como desarrollar una aplicacion web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y montado una aplicacion basica que ahora vamos a desarrollar.

Esto es lo que hemos implementado hasta ahora:

- Una pagina de inicio, **'index.jsp'**, la pagina de bienvenida de nuestra aplicacion. Fue usada para comprobar que nuestra configuracion era correcta. Mas tarde la cambiaremos para proveer un enlance a nuestra aplicacion.
- Un controlador frontal, `DispatcherServlet` con el correspondiente archivo de configuracion **'springapp-servlet.xml'** .
- Un controlador de pagina, `HelloController`, con funcionalidad limitada – simplemente devuelve un objeto `ModelAndView`. Actualmente tenemos un modelo vacio, mas tarde proveeremos un modelo completo.
- Una unidad de test para la pagina del controlador, `HelloControllerTests`, para verificar que el nombre de la vista es el que esperamos.
- Una vista, **'hello.jsp'**, que de nuevo es extremadamente sencilla. Las buenas noticias son que el conjunto de la aplicacion funciona y que estamos listos para añadir mas funcionalidad.

2.1. Configurar JSTL y añadir un archivo de cabecera JSP

Vamos a usar la Libreria Estandar JSP (JSP Standard Tag Library - JSTL), asi que comencemos copiando los archivos JSTL que necesitamos en el directorio **'WEB-INF/lib'**. Copia los archivos **jstl.jar** ubicado en el directorio **'spring-framework-2.5/lib/j2ee'** asi como **standard.jar** ubicado en el directorio **'spring-framework-2.5/lib/jakarta-taglibs'** al directorio **'springapp/war/WEB-INF/lib'**.

Vamos a crear un archivo de 'cabecera' que sera embebido en todas las paginas JSP que vamos a escribir. Asi estaremos seguros que las mismas definiciones son incluidas en todos nuestros JSP al insertar el archivo de cabecera. Tambien vamos a poner todos nuestros archivos JSP en un directorio llamado **'jsp'** bajo el directorio **'WEB-INF'**. Esto asegurara que nuestras vistas solo pueden ser accedidas a traves del controlador, por lo que no sera posible acceder a estas paginas a traves de una direccion URL. Esta estrategia podria no funcionar en algunos servidores de aplicaciones; si ese es tu caso, mueve el directorio **'jsp'** un nivel hacia arriba y usa **'springapp/war/jsp'** como el directorio de JSP en todos los ejemplos que encontraras a lo largo del tutorial, en lugar de **'springapp/war/WEB-INF/jsp'**.

Primero creamos un archivo de cabecera para incluir en todos los archivos JSP que vamos a crear.

'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" %>
```

Ahora podemos actualizar **'index.jsp'** para que incluya este archivo, y puesto que estamos usando JSTL, podemos usar la etiqueta `<c:redirect/>` para recireccionar hacia nuestro controlador frontal: `Controller`. Esto significa que todas nuestras solicitudes a **'index.jsp'** iran a traves de nuestra aplicacion. Elimina los contenidos actuales de 'index.jsp' y reemplazalos con los siguientes:

'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"/>
```

Mueve **'hello.jsp'** al directorio **'WEB-INF/jsp'**. Añade la misma directiva include que hemos añadido en **'index.jsp'** a **'hello.jsp'**. Vamos a añadir tambien la fecha y hora

actual, que seran leidas desde el modelo que pasaremos a la vista, y que mostraremos usando la etiqueta JSTL <c:out/>.

'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. Mejorar el controlador

Antes de actualizar la localizacion del JSP en nuestro controlador, actualicemos nuestra unidad de test. Sabemos que necesitamos actualizar la referencia a la vista con su nueva localizacion, `'WEB-INF/jsp/hello.jsp'`. Tambien sabemos que deberia haber un objeto en el modelo mapeado a la clave `"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);
    }
}
```

A continuacion, ejecutamos Ant con la opcion `'tests'` y nuestro test debe fallar.

```
$ 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
```

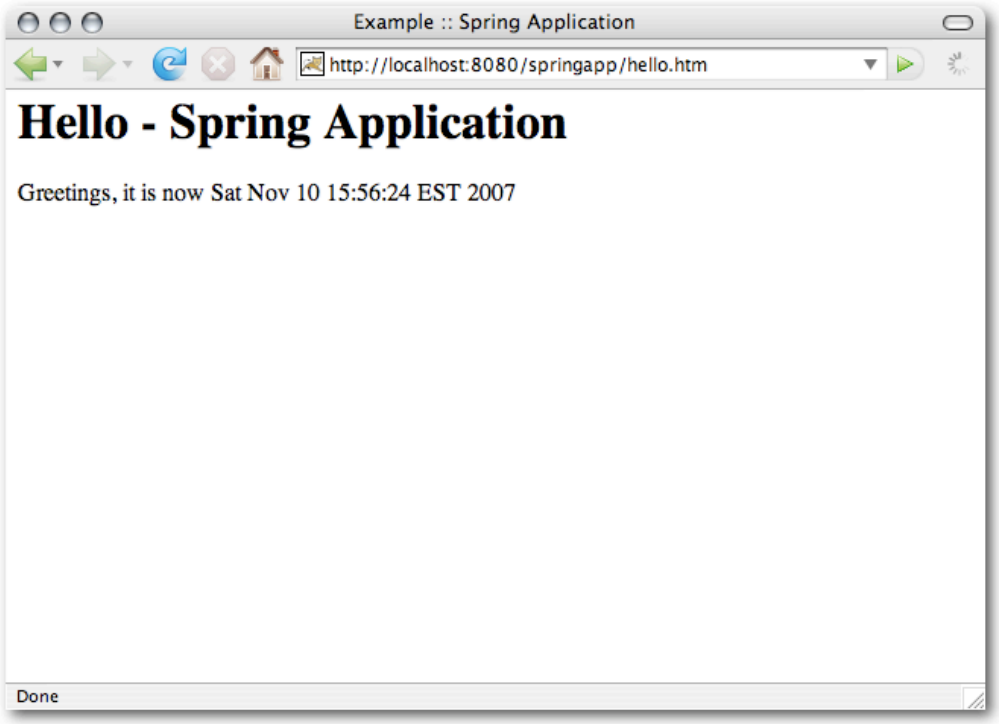
Ahora actualizamos `HelloController` configurando la referencia a la vista con su nueva localizacion, **`'WEB-INF/jsp/hello.jsp'`**, asi como la pareja clave/valor con la fecha y hora actual con la clave `"now"` y el valor: `'now'`.

`'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);
    }

}
```

Ejecutamos de nuevo Ant con la opcion `'tests'` y el test pasa.

Recuerda que el `Controller` ha sido previamente configurado en el archivo **`'springapp-servlet.xml'`** , por lo que estamos listos para probar nuestras mejoras despues de construir y desplegar el nuevo codigo. Ejecuta las tareas `'deploy reload'` en Ant, como hicimos previamente. Cuando introduzcamos la direccion http://localhost:8080/springapp/ en un navegador, deberia ejecutarse la pagina de bienvenida **`'index.jsp'`**, la cual deberia redireccionarnos a **`'hello.htm'`** que es manejada por `DispatcherServlet`, quien a su vez delega nuestra solicitud al controlador de pagina, que inserta la fecha y hora en el modelo y lo pone a disposicion de la vista **`'hello.jsp'`**.



La aplicacion actualizada

2.3. Separar la vista del controlador

Ahora el controlador especifica la ruta completa a la vista, lo cual crea una dependencia innecesaria entre el controlador y la vista. Idealmente queremos referirnos a la vista usando un nombre logico, permitiendonos intercambiar la vista sin tener que cambiar el controlador. Puedes crear este mapeo en un archivo de propiedades si estas usando `ResourceBundleViewResolver` y `SimpleUrlHandlerMapping`. Para el mapeo basico entre una vista y una localizacion, simplemente configura un prefijo y sufijo en `InternalResourceViewResolver`. Esta solucion es la que vamos a implantar ahora, por lo que modificamos **`'springapp-servlet.xml'`** y declaramos una entrada `'viewResolver'`. Eligiendo `JstlView` tendremos la oportunidad de usar JSTL en combinacion con paquetes de mensajes de idioma, los cuales nos ofreceran soporte para la internacionalizacion de la aplicacion.

`'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>
```

Actualizamos el nombre de la vista en la clase de pruebas del controlador `HelloControllerTests` por `'hello'` y relanzamos el test para comprobar que falla.

`'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);
    }
}
```

Ahora eliminamos el prefijo y sufijo del nombre de la vista en el controlador, dejando al controlador referirse a la vista por su nombre logico `"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);
    }

}
```

Relanzamos el test y ahora debe pasar.

Compilamos y desplegamos la aplicacion, y verificamos que todavia funciona.

2.4. Resumen

Echamos un vistazo rapido a lo que hemos creado en la Parte 2.

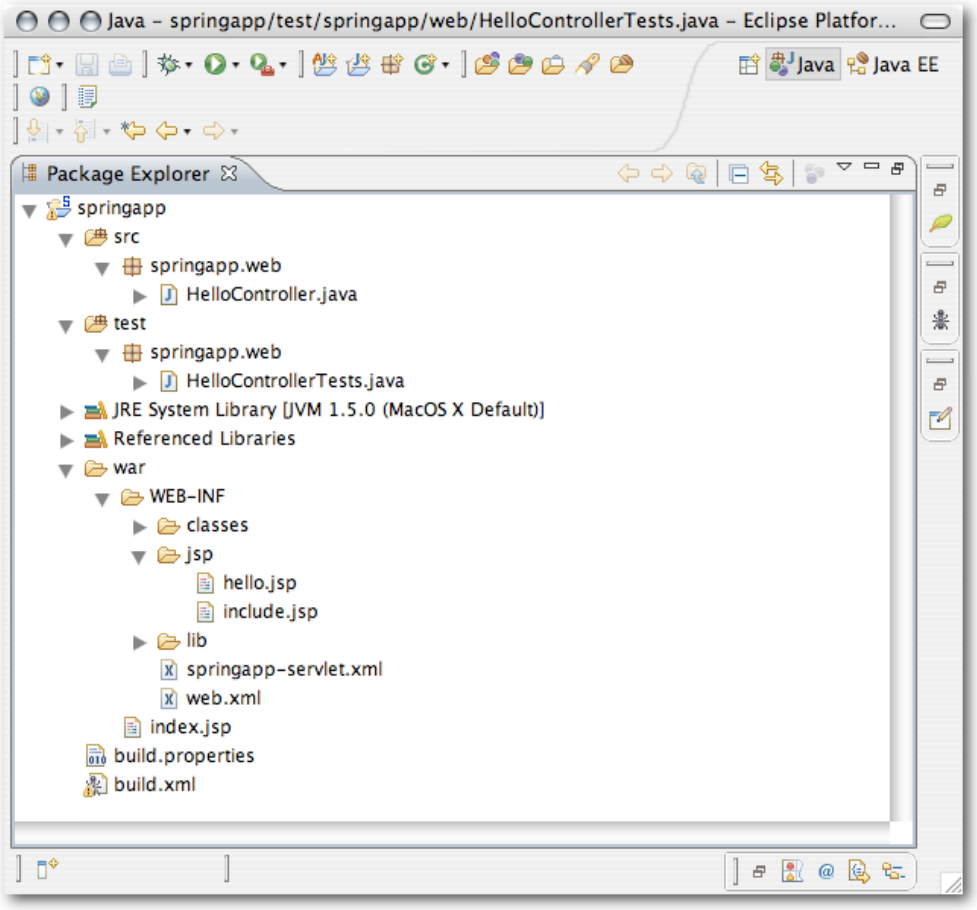
- Un archivo de cabecera **'include.jsp'**, el archivo JSP que contiene la directiva taglib que usaremos en todos nuestros archivos JSPs.

Estos son los componentes de la aplicacion que hemos cambiado en la Parte 2.

- `HelloControllerTests` ha sido actualizado repetidamente para hacer al controlador referirse al nombre logico de la vista en lugar de a su localizacion y nombre completo.

  El controlador de pagina, `HelloController`, ahora hace referencia a la vista por su nombre logico mediante el uso del `'InternalResourceViewResolver'` definido en **'springapp-servlet.xml'**.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 2

Capitulo 3. Desarrollando la Logica de Negocio

Esta es la Parte 3 del tutorial paso a paso para desarrollar una aplicacion Spring MVC. En esta seccion, adoptaremos un acercamiento pragmatico a Test-Driven Development (TDD o Desarrollo Conducido por Tests) para crear los objetos de dominio e implementar la logica de negocio para nuestro sistema de mantenimiento de inventario. Esto significa que "escribiremos un poco de codigo, lo testearemos, escribiremos un poco mas de codigo, lo volveremos a testear..." En la Parte 1 hemos configurado el entorno y montado la aplicacion basica. En la Parte 2 hemos refinado la aplicacion desacoplando la vista del controlador.

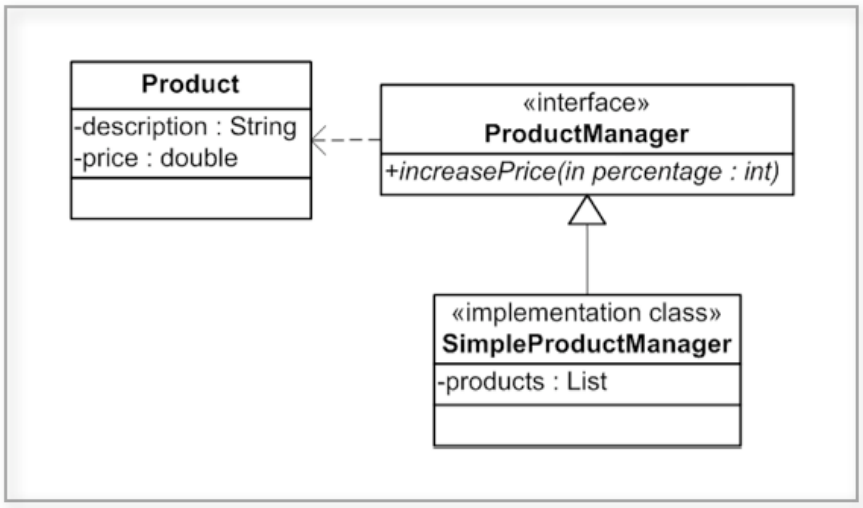
Spring permite hacer las cosas simples faciles y las dificiles posibles. La estructura fundamental que hace esto posible es el uso de `Plain Old Java Objects (POJOs u Objetos Normales Java)` por Spring. Los POJOs son esencialmente clases nomales Java libres de cualquier contrato (normalmente impuesto por un framework o arquitectura a traves de subclases o de la implementacion de interfaces). Los POJOs son objetos normales Java que estan libres de dichas obligaciones, haciendo la programacion orientada a objetos posible de nuevo. Cuando trabajes con Spring, los objetos de dominio y los servicios que implementes seran POJOs. De hecho, casi todo lo que implementes deberia ser un POJO. Si no es asi, deberias preguntarte a ti mismo porque no ocurre esto. En esta seccion, comenzaremos a ver la simplicidad y potencia de Spring.

3.1. Revisando la regla de negocio del Sistema de Mantenimiento de Inventario

En nuestro sistema de mantenimiento de inventario tenemos dos conceptos: el de producto, y el de servicio para manejarlo. Ahora el negocio solicita la capacidad de incrementar precios sobre todos los productos. Cualquier decremento sera hecho sobre productos en concreto, pero esta caracteristica esta fuera de la funcionalidad de nuestra aplicacion. Las reglas de validacion para incrementar precios son:

- El incremento maximo esta limitado al 50%.
- El incremento minimo debe ser mayor del 0%.

Debajo puedes ver un diagrama de clase para nuestro sistema de mantenimiento de inventario.



El diagrama de clase para el sistema de mantenimiento de inventario

3.2. Añadir algunas clases a la logica de negocio

Añadamos ahora mas logica de negocio en la forma de una clase `Product` y un sercicio

al que llamaremos `ProductManager` que gestionara todos los productos. Para separar la logica de la web de la logica de negocio, colocaremos las clases relacionadas con la capa web en el paquete `'web'` y crearemos dos nuevos paquetes: uno para los objetos de servicio, al que llamaremos `'service'`, y otro para los objetos de dominio al que llamaremos `'domain'`.

Primero implementamos la clase `Product` como un POJO con un constructor por defecto (que es provisto si no especificamos ningun constructor explicitamente), asi como metodos getters y setters para las propiedades `'description'` y `'price'`. Ademas haremos que la clase implemente la interfaz `Serializable`, no necesariamente para nuestra aplicacion, pero que sera necesario mas tarde cuando persistamos y almacenemos su estado. Esta clase es un objeto de dominio, por lo tanto pertenece al paquete `'domain'`.

`'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();
    }
}
```

Escribamos ahora una unidad de test para nuestra clase `Product`. Algunos programadores no se molestan en escribir tests para los getters y setters, tambien llamado codigo 'auto-generado'. Normalmente conlleva mucho tiempo retirarse del debate (como este parrafo demuestra) sobre si los getters and setters necesitan ser testeados, ya que son metodos demasiado 'triviales'. Nosotros escribiremos los tests debido a: a) son triviales de escribir; b) tenemos siempre los tests pagando dividendos en terminos de tiempo salvado si en solo una ocasion de cien nos vemos salvados de un error producido por un getter o setter; y c) porque mejoran la cobertura de los tests. Creamos un stub de `Product` y testeamos cada metodo getter y setter como un pareja en un test simple. Normalmente, escribiras uno o mas metodos de test por cada metodo de la clase, con cada metodo de test comprobando una condicion particular en el metodo de la clase (como verificar un valor `null` pasado al metodo).

`'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);
    }

}
```

A continuacion creamos `ProductManager`. Este es el servicio responsable de manejar productos. Contiene dos metodos: un metodo de negocio, `increasePrice()`, que incrementa el precio de todos los productos, y un metodo getter, `getProducts()`, para recuperar todos los productos. Hemos decidido diseñarlo como una interface en lugar de como una clase concreta por algunas razones. Primero, es mas facil escribir tests de unidad para `Controllers` (como veremos en el proximo capitulo). Segundo, el uso de interfaces implica que JDK Proxying (una caracteristica del lenguaje Java) puede ser usada para hacer el servicio transaccional, en lugar de usar CGLIB (una libreria de generacion de codigo).

'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();

}
```

Vamos a crear ahora la clase `SimpleProductManager` que implementa la interface `ProductManager`.

'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();
    }

}
```

Antes de implementar los metodos en `SimpleProductManager`, vamos a definir algunos tests. La definicion mas estricta de `Test Driven Development` (TDD) implica escribir siempre los tests primero, y a continuacion el codigo. Una interpretacion aproximada seria mas parecido a `Test Oriented Development` (TOD - Desarrollo Orientado a Tests), donde alternariamos entre escribir el codigo y los tests como parte del proceso de desarrollo. Lo mas importante es tener para el codigo base el conjunto mas completo de tests que sea posible, de manera que la forma en que alcances este objetivo es mas teoria que practica. Muchos programadores TDD, sin embargo, estan de acuerdo en que la calidad de los tests es siempre mayor cuando son escritos al mismo tiempo que el codigo, por lo que esta es la aproximacion que vamos a tomar.

Para escribir test efectivos, tienes que considerar todas las pre- y post-condiciones del metodo que va a ser testeado, asi como lo que ocurre dentro del metodo. Comencemos testeando una llamada a `getProducts()` que devuelve `null`.

'springapp/test/springapp/service/SimpleProductManagerTests.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());
    }

}
```

Relanza Ant con la opcion `tests` y el test debe fallar, ya que `getProducts()` todavia no ha sido implementado. Normalmente es una buena idea marcar los metodos aun no implementados haciendo que lancen una excepcion de tipo `UnsupportedOperationException`.

A continuacion vamos a implementar un test para recuperar una lista de objectos de respaldo en los que han sido almacenados datos de prueba. Sabemos que tenemos que almacenar la lista de productos en la mayoria de nuestros tests de `SimpleProductManagerTests`, por lo que definimos la lista de objetos de respaldo en el metodo `setUp()` de JUnit, el cual es invocado previamente a cada llamada a un metodo de test.

'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);

    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());
    }
}
```

Relanza Ant con la opcion `tests` y nuestros dos tests deben fallar.

Volvemos a `SimpleProductManager` e implementamos ambos metodos getter and setter para la propiedad `products`.

'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;
    }

}
```

Relanza Ant con la opcion `tests` y ahora todos los tests deben pasar.

Ahora procedemos a implementar los siguientes test para el metodo `increasePrice()`:

- La lista de productos es null y el metodo se ejecuta correctamente.
- La lista de productos esta vacia y el metodo se ejecuta correctamente.
- Fija un incremento de precio del 10% y comprueba que dicho incremento se ve reflejado en los precios de todos los productos de la lista.

'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());
    }
```

```
}
```

Volvemos a `SimpleProductManager` para implementar `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;
    }

}
```

Relanzamos Ant con la opcion `tests` y todos nuestros tests deben pasar. ¡HURRA!. JUnit tiene un dicho: "keep the bar green to keep the code clean (manten la barra verde para mantener el codigo limpio)". Esto es debido a que los IDE con soporte para JUnit utilizan una barra de color verde para indicar que todos los tests han pasado, y una de color morado para indicar que han habido fallos. Para aquellos que estais ejecutando los tests en un IDE y sois nuevos en tests de unidad, esperamos que os sintais invadidos por una gran sensacion de seguridad y confianza al saber que el codigo es verdaderamente operativo como esta especificado en las reglas de negocio que habeis definido previamente. Nosotros realmente lo estamos.

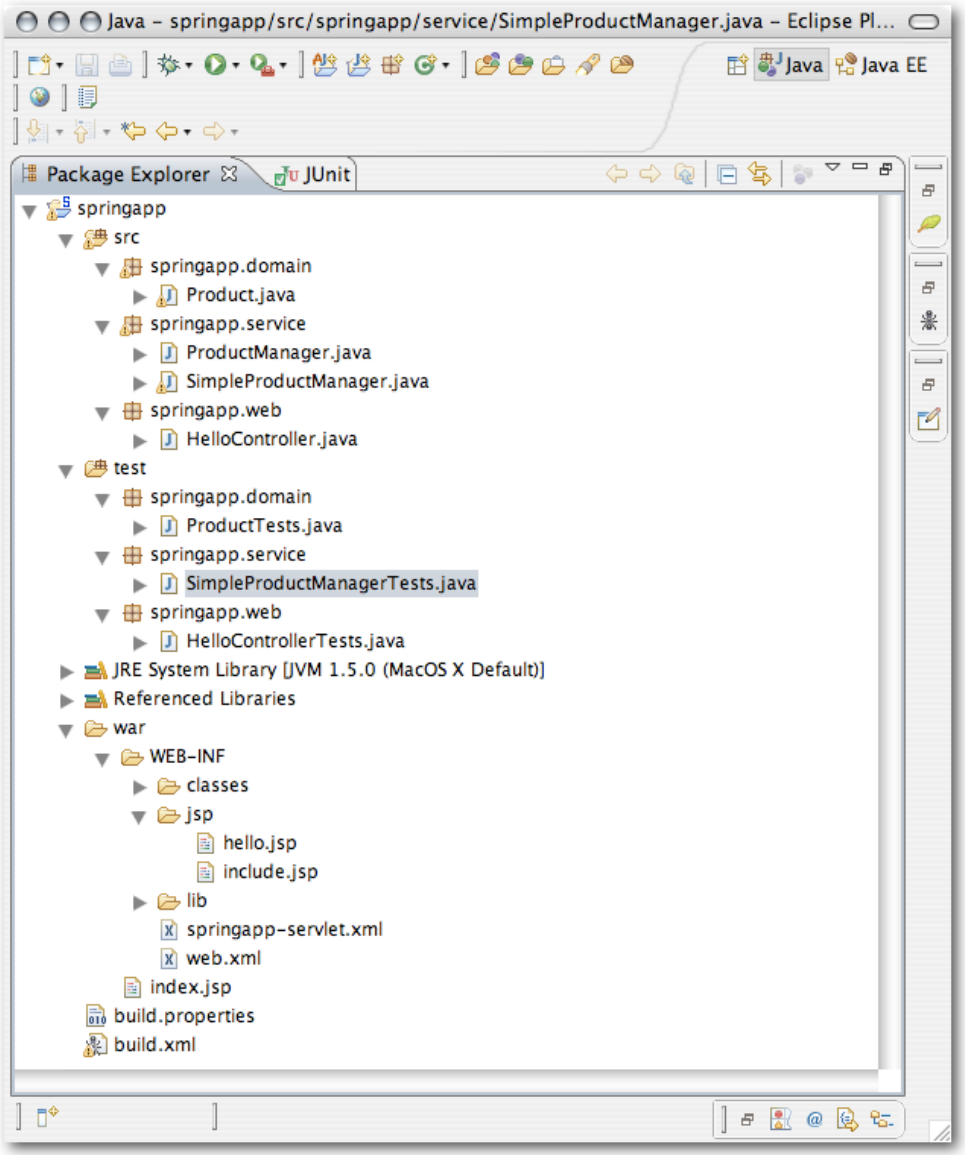
Ahora estamos listos para movernos a la capa web para poner una lista de productos en nuestro modelo `Controller`.

3.3. Resumen

Echemos un rapido vistazo a lo que hemos hecho en la Parte 3.

- Hemos implementado el objeto de dominio `Product` , la interface de servicio `ProductManager` y la clase concreta `SimpleProductManager` , todos como POJOs.
- Hemos escrito tests de unidad para todas las clases que hemos implementado.
- No hemos escrito ni una sola linea de codigo de Spring. Este es el ejemplo de lo no-intrusivo que es realmente Spring Framework. Uno de sus propositos principales es permitir a los programadores centrarse en la parte mas importante de todas: crear valor modelando e implementando requerimientos de negocio. Otro de sus propositos es hacer seguir las mejores practicas de programacion mas faciles, como implementar servicios usando interfaces y usando tests de unidad, mas alla de las obligaciones pragmaticas de un proyecto dado. A lo largo de este tutorial, veras como los beneficios de diseñar interfaces cobran vida.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 3

Capitulo 4. Desarrollando la Interface Web

Esta es la Parte 4 del tutorial paso a paso para desarrollar una aplicacion web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y montado la aplicacion basica. En la Parte 2 hemos mejorado la aplicacion que habiamos construido hasta entonces. La Parte 3 añade toda la logica de negocio y los tests de unidad. Ahora es el momento de construir la interface web para la aplicacion.

4.1. Añadir una referencia a la logica de negocio en el controlador

Para empezar, renombramos `HelloController` a algo mas descriptivo, como por ejemplo `InventoryController`, puesto que estamos construyendo un sistema de inventario. Aqui es donde un IDE con opcion de refactorizar es de valor incalculable. Renombramos `HelloController` a `InventoryController` asi como `HelloControllerTests` a `InventoryControllerTests`. A continuacion, ,modificamos `InventoryController` para almacenar una referencia a la clase `ProductManager`. Tambien añadimos codigo para permitir al controlador pasar algo de informacion sobre un producto a la vista. El metodo `getModelAndView()` ahora devuelve tanto un `Map` con la fecha y hora como una lista de productos.

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

```
package springapp.web;

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

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

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

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

import springapp.service.ProductManager;

public class InventoryController implements Controller {

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

    private ProductManager productManager;

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

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

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

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


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

}
```

Tambien necesitaremos modificar InventoryControllerTest para proporcionar un ProductManager y extraer el valor para `'now'` desde el modelo Map antes de que los test sean pasados de nuevo.

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

```
package springapp.web;

import java.util.Map;

import org.springframework.web.servlet.ModelAndView;

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

import junit.framework.TestCase;

public class InventoryControllerTests extends TestCase {

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

4.2. Modificar la vista para mostrar datos de negocio y añadir soporte para archivos de mensajes

Usando la etiqueta JSTL `<c:forEach/>`, añadimos una seccion que muestra informacion de cada producto. Tambien vamos a reemplazar el titulo, la cabecera y el texto de bienvenida con una etiqueta JSTL `<fmt:message/>` que extrae el texto a mostrar desde una ubicacion `'message'` – veremos esta ubicacion un poco mas adelante.

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

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

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

4.3. Añadir datos de prueba para rellenar algunos objetos de negocio

Es el momento de añadir `SimpleProductManager` a nuestro archivo de configuracion y de pasarlo a traves del setter de `InventoryController`. Todavia no vamos a añadir ningun codigo para cargar los objetos de negocio desde una base de datos. En su lugar, podemos reemplazarlos con unas cuantas instancias de la clase `Product` usando beans Spring y el soporte de la aplicacion. Simplemente pondremos los datos que necesitamos en un puñado de entradas bean en el archivo `'springapp-servlet.xml'`. Tambien añadiremos la entrada bean para `'messageSource'` que nos permitira recuperar mensajes desde la ubicacion (`'messages.properties'`) que crearemos en el proximo paso. Ademas, debes renombrar la referencia a `HelloController` por `InventoryController` puesto que le hemos cambiado el nombre.

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

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

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

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

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

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

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

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

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

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

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

</beans>
```

4.4. Añadir una ubicacion para los mensajes y la tarea `'clean'` a `'build.xml'`

Creamos un archivo llamado `'messages.properties'` en el directorio `'war/WEB-INF/classes'`. Este archivo de propiedades contiene tres entradas que coinciden con las claves especificadas en las etiquetas `<fmt:message/>` que hemos añadido a `'hello.jsp'`.

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

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

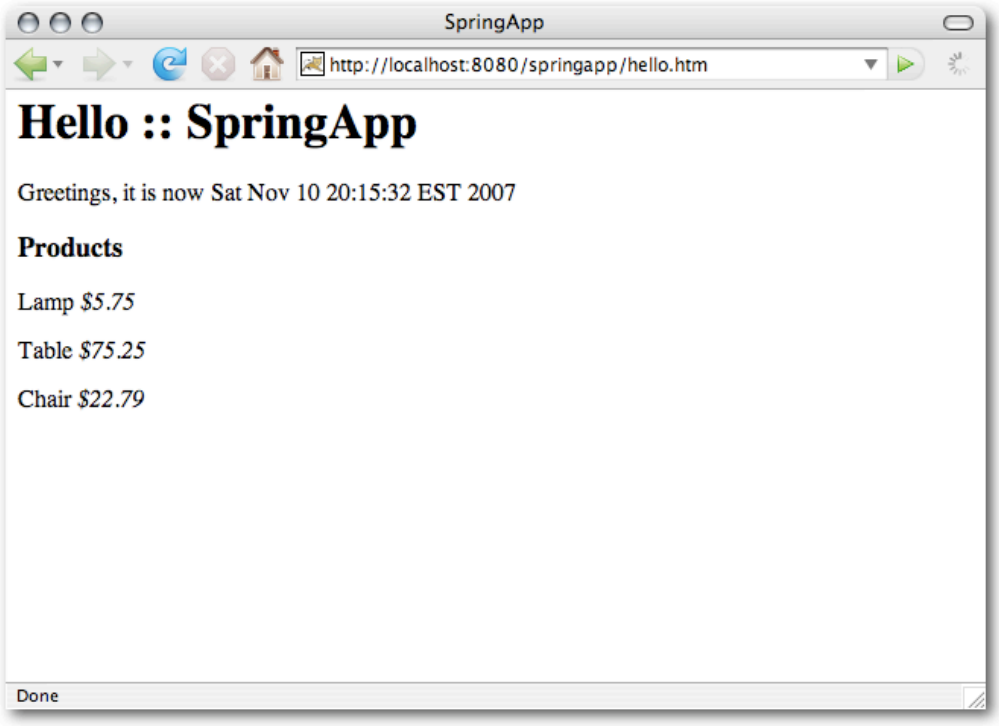
Puesto que hemos movido algunos archivos de codigo fuente, tiene sentido añadir los comandos `'clean'` y `'undeploy'` al script de Ant. Añadimos las siguientes entradas en el archivo `'build.xml'`.

`'build.xml'`:

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

    <target name="undeploy" description="Un-Deploy application">
        <delete>
            <fileset dir="${deploy.path}/${name}">
                <include name="**/*.*"/>
            </fileset>
        </delete>
    </target>
```

Ahora deten el servidor Tomcat y ejecuta Ant con las opciones `'clean'`, `'undeploy'` y `'deploy'`. Esto eliminara todos los archivos de clases, reconstruira la aplicacion y la

desplegara de nuevo. Arranca Tomcat de nuevo y deberias ver lo siguiente al cargar la aplicacion desde tu navegador:



La aplicacion actualizada

4.5. Añadir un formulario

Para proveer de una interface a la aplicacion web que muestre la funcionalidad para incrementar los precios, vamos a añadir un formulario que permitira al usuario introducir un valor de porcentaje. Este formulario usa una libreria de etiquetas llamado `'spring-form.tld'` que es suministrada con Spring Framework. Tenemos que copiar este archivo desde nuestra distribucion de Spring (**`'spring-framework-2.5/dist/resources/spring-form.tld'`**) al directorio **`'springapp/war/WEB-INF/tld'`** que ademas debemos crear. A continuacion debemos añadir tambien una entrada `<taglib/>` en el archivo **`'web.xml'`**.

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

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

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

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

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

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

  <jsp-config>
    <taglib>
```

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

</web-app>
```

Tambien tenemos que declarar este taglib en una directiva page en el siguiente archivo JSP, y ya podremos comenzar a utilizar las etiquetas que habremos asi importado. Añade el archivo JSP **'priceincrease.jsp'** al directorio **'war/WEB-INF/jsp'**.

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

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

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

La siguiente clase es un JavaBean muy sencilla que solamente contiene una propiedad con su correspondientes metodos getter y setter. Este es el objeto que el formulario rellenara y desde el que nuestra logica de negocio extraera el procentaje de incremento que queremos aplicar a los precios.

'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;
    }

}
```

La siguiente clase de validacion toma el control despues de que el usuario pulse el boton submit. Los valores introducidos en el formulario seran guardados en el objeto de comando por el framework. El metodo `validate(..)` es llamado en el objeto de comando `PriceIncrease`. Ademas un objeto que contiene cualquier error que se haya producido al completar el formulario es pasado tambien.

'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. Añadir un controlador de formulario

Ahora tenemos que añadir una entrada en el archivo **'springapp-servlet.xml'** para definir el nuevo formulario y su controlador. Definimos los objetos a inyectar en el controlador del formulario mediante las propiedades `commandClass` y `validator`. Tambien especificamos dos vistas, `formView` que es usada por el formulario, y `successView`, a la que iremos despues de procesar satisfactoriamente el formulario. La ultima vista puede ser de dos tipos. Puede ser una referencia a una vista normal, que es enviada a uno de nuestras paginas JSP. Una desventaja de esta aproximacion es que si el usuario recarga la pagina, los datos del formulario se enviaran de nuevo, y podrias terminar incrementando varias el porcentaje. Una manera alternativa es usar redirect, donde la respuesta se envia de vuelta al navegador del usuario informandole que debe redireccionar a una nueva URL. Esta URL no puede ser una de nuestras paginas JSP, puesto que estan ocultas (en WEB-INF) y no es posible su acceso directo. Tiene que ser una URL alcanzable desde el exterior. Hemos elegido usar **'hello.htm'** como URL para redirect. Esta URL esta mapeada a la pagina **'hello.jsp'**, que funcionara correctamente.

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

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

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

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

<beans>

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

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

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

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

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

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

    <bean name="/priceincrease.htm" class="springapp.web.PriceIncreaseFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="priceIncrease"/>
        <property name="commandClass" value="springapp.service.PriceIncrease"/>
        <property name="validator">
            <bean class="springapp.service.PriceIncreaseValidator"/>
        </property>
        <property name="formView" value="priceincrease"/>
        <property name="successView" value="hello.htm"/>
        <property name="productManager" ref="productManager"/>
    </bean>

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

</beans>
```

A continuacion, echemos un vistazo al controlador de este formulario. El metodo `onSubmit(..)` toma el control y hace algo de logging antes de llamar al metodo `increasePrice(..)` en el objeto `ProductManager`. Entondes devuelve un objeto `ModelAndView` pasando en el una nueva instancia de `RedirectView` que es creada usando la URL de la vista que se mostrara si no hay ningun error en el formulario.

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

```
package springapp.web;

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

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

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

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

public class PriceIncreaseFormController extends SimpleFormController {

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

    private ProductManager productManager;

    public ModelAndView onSubmit(Object command)
            throws ServletException {

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

        productManager.increasePrice(increase);

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

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

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

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

    public ProductManager getProductManager() {
        return productManager;
    }

}
```

Vamos a añadir tambien algunos mensajes al archivo de mensajes **'messages.properties'**.

'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!!!
```

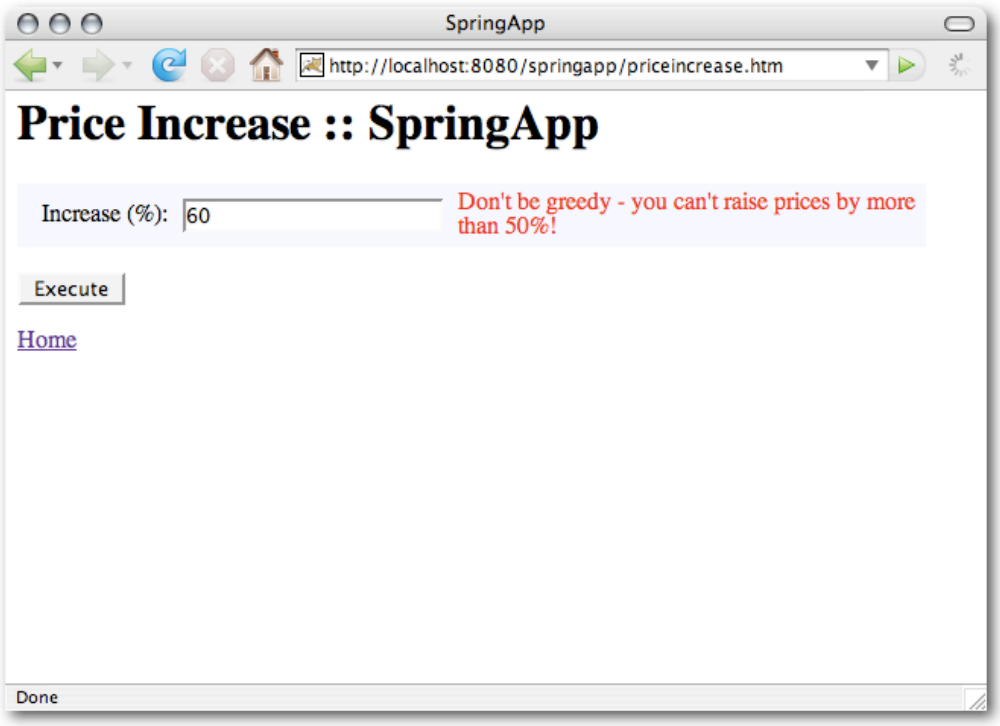
Compila y despliega, y despues de recargar la aplicacion podemos probarla. Ahora el formulario puede mostrar los errores.

Finalmente, vamos a añadir un enlace a la pagina de incremento de precio desde **'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>
```

Ahora, ejecuta Ant con los comandos 'deploy' y 'reload' y prueba la nueva funcionalidad de incremento de precio.



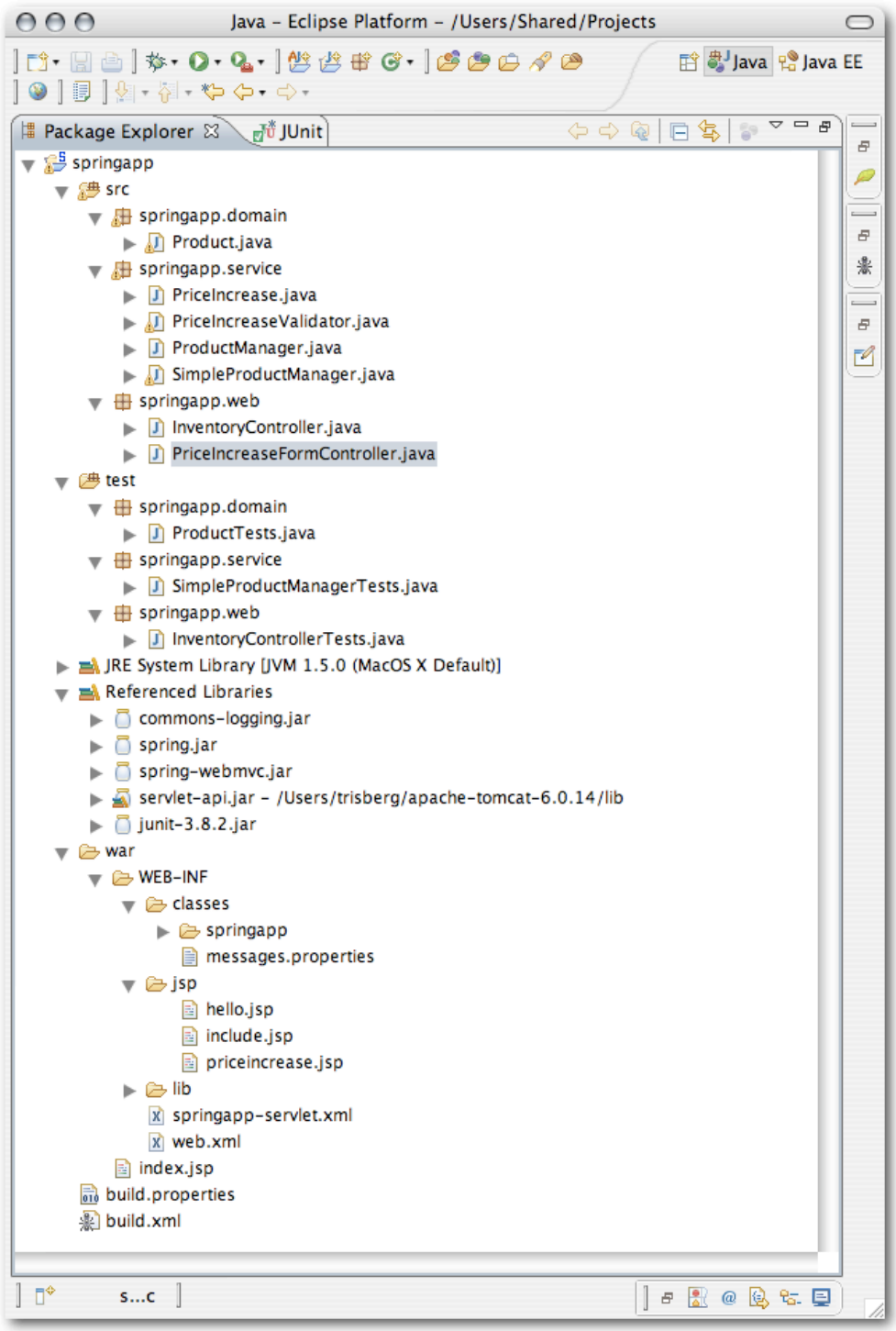
La aplicacion actualizada

4.7. Resumen

Vamos a ver lo que hemos hecho en la Parte 4.

- Hemos renombrado nuestro controlador a `InventoryController` y le hemos dado una referencia a `ProductManager` por lo que ahora podemos recupear una lista de productos para mostrar.
- Entonces hemos definido algunos datos de prueba para rellenar objetos de negocio.
- A continuacion hemos modificado la pagina JSP para usar una ubicacion de mensajes y hemos añadido un loop forEach para mostrar una lista dinamica de productos.
- Despues hemos creado un formulario para disponer de la capacidad de incrementar los precios.
- Finalmente hemos creado un controlador de formulario y un validador, y hemos desplegado y probado las nuevas caracteristicas.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 4

Capitulo 5. Implementando Persistencia en Base de Datos

Esta es la Parte 5 del tutorial paso a paso sobre sobre como desarrollar una aplicacion web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y puesto en marcha una aplicacion basica. En la Parte 2 hemos mejorado la aplicacion que habiamos construido hasta entonces. En la Parte 3 hemos añadido toda la logica de negocio y los test de unidad, y en la Parte 4 desarrollado la interface web. Ahora es el momento de introducir persistencia en base de datos. En las partes anteriores hemos visto como cargar algunos objetos de negocio definiendo beans en un archivo de configuracion. Es obvio que esta solucion nunca va a funcionar en el mundo real – cada vez que reiniciemos el servidor obtendremos de nuevo los precios originales. Necesitamos añadir codigo para persistir esos cambios en una base de datos.

5.1. Crear un script de inicio de base de datos

Antes de que podamos comenzar a desarrollar el codigo de persistencia, necesitamos una base de datos. Hemos planeado usar HSQL, la cual es una buena base de datos escrita en Java. Esta base de datos es distribuida con Spring por lo que podemos copiar su archivo jar al directorio lib de la aplicacion web. Copia el archivo `hsqldb.jar` del directorio `'spring-framework-2.5/lib/hsqldb'` al directorio `'springapp/war/WEB-INF/lib'`. Vamos a usar HSQL en modo standalone. Esto significa que tendremos que arrancar la base de datos de forma serparada en lugar de confiar en una base de datos integrada con la propia aplicacion. De esta manera sera mas sencillo ver los cambios hechos en la base de datos cuando ejecutemos la aplicacion.

Necesitamos un script o un archivo de lotes para iniciar la base de datos. Crea un directorio `'db'` dentro del directorio principal `'springapp'`. Este nuevo directorio contendra los archivos de la base de datos. Ahora, añadamos un script de inicio:

Desde Linux/Mac OS X añadimos a:

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

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

No olvides cambiar los permisos de ejecucion con el comando **'chmod +x server.sh'**.

Desde Windows añadimos a:

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

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

Ahora ya puedes abrir una ventana de comandos, ir al directorio `'springapp/db'` y arrancar la base de datos ejecutando el script de arranque correspondiente a tu sistema operativo.

5.2. Crear una tabla y scripts de prueba de datos

Primero, vamos a ver las sentencias SQL necesarias para crear la tabla. Crea el archivo 'create_products.sql' en el directorio db.

`'springapp/db/create_products.sql' con el siguiente contenido`:

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

Ahora necesitamos añadir nuestros datos de prueba. Crea el archivo 'load_data.sql' en el directorio db con el siguiente contenido.

`'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);
```

En la seccion siguiente vamos a añadir algunas tareas Ant a su script de construccion de

manera que podamos ejecutar los scripts SQL.

5.3. Añadir tareas Ant para ejecutar los scripts SQL y cargar datos de prueba

Vamos a crear tablas y poblarlas con datos de prueba usando el comando incorporado en Ant "sql". Para usarlo necesitamos añadir algunos parametros de conexion a la base de datos en un archivo de propiedades

'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=
```

A continuacion añadimos los tareas que necesitamos al archivo de contruccion de Ant. Hay tareas para crear y borrar tablas, y para cargar y borrar datos de prueba.

Añade las siguientes tareas a '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>
```

Ahora puedes ejecutar **'ant createTables loadData printData'** para preparar los datos de prueba que vamos a usar despues.

5.4. Crear una implementacion para JDBC de un Objeto de Acceso a Datos (DAO)

Comencemos creando un nuevo directorio llamado `'src/springapp/repository'` que contendra cualquier clase que sea usada para el acceso a la base de datos. En este directorio vamos a crear un nuevo interface llamado `ProductDao`. Este sera el interface que definira la funcionalidad de la implementacion DAO que vamos a crear - esto nos permitira elegir en el futuro otra implementacion que se adapte mejor a nuestras necesidades.

'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);

}
```

A continuacion creamos una clase llamada `JdbcProductDao` que sera la implementacion JDBC de la interface anterior. Spring dispone de un framework de abstraccion JDBC que vamos a usar. La mayor diferencia entre usar JDBC directamente y el framework JDBC de Spring es que no tienes que preocuparte de abrir o cerrar conexiones, o cualquier codigo similar. Todo esto es manejado de manera automatica. Otra ventaja es que no tienes que capturar ninguna excepcion, a menos que quieras. Spring envuelve todas las excepciones de tipo SQLException en un familia de excepciones de tipo unchecked que heredan de DataAccessException. Si lo deseas, puedes capturar esta excepcion, pero puesto que muchas excepciones de base de datos son imposibles de recuperar de ninguna manera, puedes simplemente dejar que esta excepcion se propague hacia un nivel superior. La clase `SimpleJdbcDaoSupport` provee el acceso necesario para obtener un previamente configurado objeto `SimpleJdbcTemplate`, por lo que podemos heredar de

esta clase. Todo lo que tenemos que proveer en el contexto de la aplicacion es un `DataSource` convenientemente configurado.

'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;
        }

    }

}
```

Vamos a echarle un vistazo a los dos metodos DAO en esta clase. Puesto que estamos extendiendo `SimpleJdbcSupport` disponemos de un objeto `SimpleJdbcTemplate` preparado y listo para usar. Este objeto es accedido llamando al metodo `getSimpleJdbcTemplate()`.

El primer metodo, `getProductList()` ejecuta una consulta usando `SimpleJdbcTemplate`. Para ello incluimos en el una sentencia SQL y una clase que pueda manejar el mapeo entre el el `ResultSet` y la clase `Product`. En nuestro caso este mapeador es una clase llamada `ProductMapper` que hemos definido como una clase interna del DAO. Por supuesto que esta clase no sera usada fuera del DAO por lo que hacerla interna es una buena solucion.

ProductMapper implementa la interface `ParameterizedRowMapper` que define un unico metodo llamado `mapRow` , y que por tanto debe ser implementado. Este metodo mapeara los datos de cada fila de la base de datos a una clase que representa la entidad que estas recuperando con tu consulta. Puesto que `RowMapper` es parametizado, el metodo `mapRow` devuelve el mismo tipo que ha creado.

El segundo metodo, saveProduct, tambien usa `SimplJdbcTemplate`. Esta vez hacemos un `update` pasando la correspondiente sentencia SQL junto con el valor de los parametros mediante un objeto `MapSqlParameterSource`. Usar `MapSqlParameterSource` nos permite

usad parametros con nombre en lugar de los caracteres "?" que hubieras necesitado para escribir una sentencia SQL. Los parametros con nombre hacen tu codigo mas explicito y evitan problemas causados por parametros con valores incorrectos (debido a errores de ordenacion, etc). El metodo update devuelve el numero de filas afectadas en la base de datos.

Necesitamos almacenar el valor de la primera clave para cada producto de la clase Product. Esta clave sera usada cuando realicemos cualquier cambio en el objeto y lo volvamos a persistir en la base de datos. Para almacenar esta clave añadimos una variable privada llamada 'id' complementada con sus correspondientes getters y setters.

'springapp/src/springapp/domain/Product.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();
    }
}
```

Esto completa la implementacion JDBC en nuestra capa de persistencia.

5.5. Implementar tests para la implementacion DAO sobre JDBC

Es el momento de añadir tests a nuestra aplicacion DAO sobre JDBC. Spring dispone de un extenso framework para tests que soporta JUnit 3.8 y 4 asi como TestNG. No podemos cubrir todos ellos en esta guia pero vamos a mostrar una implementacion simple basada en el soporte especifico para JUnit 3.8 de Spring. Necesitamos añadir a nuestro proyecto el archivo jar que contiene el framework de tests de Spring. Copia **spring-test.jar** desde el directorio **'spring-framework-2.5/dist/modules'** hasta el directorio **'springapp/war/WEB-INF/lib'**.

Ahora podemos crear nuestra clase de tests. Extendiendo `AbstractTransactionalDataSourceSpringContextTests` obtenemos un monton de caracteristicas muy utiles de manera totalmente automatica. Conseguimos inyeccion de dependencias desde el contexto de la aplicacion en cualquier metodo setter. Este contexto de aplicacion es cargado por el framework de tests. Todo lo que necesitamos hacer es especificar su nombre en el metodo `getConfigLocations`. Ademas, obtenemos la oportunidad de preparar nuestra base de datos con los datos de prueba apropiados a traves del metodo `onSetUpInTransaction`. Esto es importante, puesto que desconocemos el estado de la base de datos cuando ejecutamos nuestros tests. Puesto que estamos extendiendo un test "Transactional", cualquier cambio que hagamos sera

automaticamente cancelado una vez que el test finalice. Los metodos `deleteFromTables` y `executeSqlScript` estan definidos en la superclase, por lo que no tenemos que implementarlos para cada test. Simplemente hay que pasarle los nombres de tabla a vaciar y el nombre del script que contiene los datos de prueba.

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

```
package springapp.repository;

import java.util.List;

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());
        }

    }

}
```

Aun no disponemos del archivo que contiene el contexto de la aplicacion, y que es cargado por este test, por lo que vamos a crear este archivo en el directorio **'springapp/test'**:

'springapp/test/test-context.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>
```

Hemos definido un `productDao` el cual es la clase que estamos testeando. Ademas hemos definido un `DataSource` con comodines para los valores de configuracion. Sus valores seran ajustados mediante un archivo de propiedades en tiempo de ejecucion. La clase `PropertyPlaceholderConfigurer` que hemos declarado leera este archivo de propiedades y sustituira cada comodin con su valor actual. Esto es conveniente puesto que separa los valores de conexion en su propio archivo, y estos valores a menudo suelen ser cambiados durante el despliegue de la aplicacion. Vamos a poner este nuevo archivo en el directorio **`'war/WEB-INF/classes'`** por lo que estara disponible cuando ejecutemos la aplicacion ademas de cuando despleguemos la aplicacion web. El contenido de este archivo de propiedades es:

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

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

Puesto que hemos añadido un archivo de configuracion en el directorio **`'test'`**" y el archivo de propiedades jdbc.properties en el directorio **`'WEB-INF/classes'`**, vamos a añadir una nueva entrada al classpath para nuestros tests. Esta entrada deberia ir despues de la declaracion de la propiedad `'test.dir'`:

`'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>

...
```

Ahora disponemos del codigo suficiente para ejecutar nuestros tests y hacerlos pasar pero queremos hacer un cambio adicional al script de Ant. Es una buena practica separar cualquier test de integracion que depende de una base de datos real del resto de los tests. Por ello vamos a añadir una tarea alternativa llamada "dbTests" a nuestro script de Ant, y vamos a excluir los tests sobre la base de datos de la tarea "tests".

`'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>

...
```

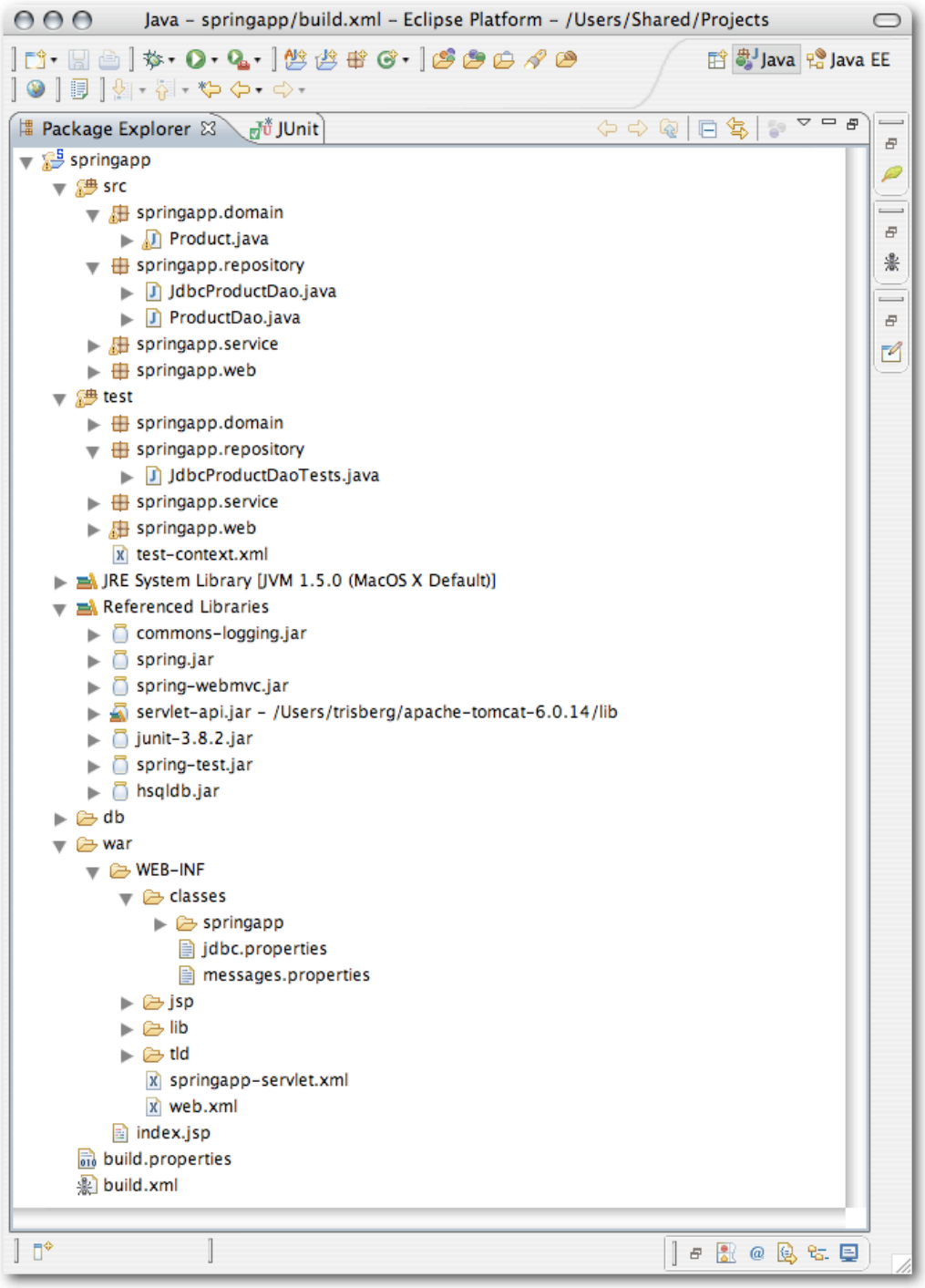
Hora de ejecutar este test, ejecuta **'ant dbTests'** para ver si el test pasa.

5.6. Resumen

Ya hemos completado la capa de persistencia y en la proxima parte vamos a integrarla con nuestra aplicacion web. Pero primero, resumamos rapidamente todo lo que hemos hecho en esta parte.

- Primero hemos configurado nuestra base de datos y creado los scripts de arranque.
- Hemos creado scripts para crear una tabla en la base de datos y cargar algunos datos de prueba.
- A continuacion hemos añadido algunas tareas a nuestro script de Ant que ejecutaremos cuando necesitemos crear o borrar la tabla, y tambien cuando necesitamos añadir y borrar los datos de prueba.
- Hemos creado una clase DAO que manejara el trabajo de persistencia usando la clase `SimpeJdbcTemplate` de Spring.
- Finalmente hemos creado tests de integracion y sus correspondientes tareas Ant para ejecutarlos.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 5

Capitulo 6. Integrando la Aplicacion Web con la Capa de Persistencia

Esta es la Parte 6 del tutorial paso a paso sobre como desarrollar una aplicacion web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y puesto en marcha una aplicacion basica. En la Parte 2 hemos mejorado la aplicacion que habiamos construido hasta entonces. En la Parte 3 hemos añadido toda la logica de negocio y los test de unidad, y en la Parte 4 desarrollado la interface web. En la Parte 5 hemos desarrollado la capa de persistencia. Ahora es el momento de integrarlo todo junto en una aplicacion web completa.

6.1. Modificar la Capa de Servicio

Si hemos estructurado nuestra aplicacion adecuadamente, solo tenemos que cambiar la capa de servicio para que haga uso de la persistencia en base de datos. Las clases de la vista y el controlador no tienen que ser modificadas, puesto que no deberian ser conscientes de ningun detalle de la implementacion de la capa de servicio. Asi que vamos a añadir persistencia a la implementacion de ProductManager. Modifica la clase `SimpleProductManager` y añade una referencia a la interface `ProductDao` ademas de un metodo setter para esta referencia. Que implementacion usemos debe ser irrelevante para la clase ProductManager, y podemos hacerlo mediante configuracion. Tambien vamos a cambiar el metodo `setProducts` a uno llamado `setProductDao` para que podamos inyectar una instancia de la clase DAO. El metodo `getProducts` usara ahora este DAO para recuperar la lista de productos. Finalmente, el metodo `increasePrices` recuperara la lista de productos y, despues de haber incrementado los precios, almacenara los productos de nuevo en la base de datos usando el metodo `saveProduct` definido en el DAO.

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

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    privapackage 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. Resolver los tests fallidos

Hemos modificado `SimpleProductManager` y ahora evidentemente los tests fallan. Necesitamos proporcionar a `ProductManager` una implementacion en memoria de `ProductDao`. Realmente no queremos usar el verdadero DAO puesto que queremos evitar tener acceso a la base de datos por nuestros tests de unidad. Añadiremos una clase llamada `InMemoryProductDao` que almacenara una lista de productos que seran definidos en el constructor. Esta clase en memoria tiene que ser pasada a `SimpleProductManager` en el momento de ejecutar los tests.

'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) {
    }

}
```

Y aqui esta la version modificada de `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();
        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());
    }

}
```

Tambien necesitamos modificar `InventoryControllerTests` puesto que esta clase tambien usa `SimpleProductManager`. Aqui esta la version modificada de `InventoryControllerTests`:

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

```
package springapp.web;

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. Crear un nuevo contexto de aplicacion para configurar la capa de servicio

Hemos visto antes que es tremendamente facil modificar la capa de servicio para usar persistencia en base de datos. Esto es asi porque esta despegada de la capa web. Ahora es el momento de despegar tambien la configuracion de la capa de servicio de la capa web. Eliminaremos la configuracion de productManager y la lista de productos del archivo de configuracion springapp-servlet.xml. Asi es como este archivo quedaria ahora:

'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>

    <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>
```

Todavia necesitamos configurar la capa de servicio y lo haremos en nuestro propio archivo de contexto de aplicacion. Este archivo se llama **'applicationContext.xml'** y sera cargado mediante un servlet listener que definiremos en **'web.xml'**. Todos los bean

configurados en este nuevo contexto de aplicacion estaran disponibles desde cualquier contexto del servlet.

'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" >

  <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>
```

Ahora creamos un nuevo archivo **'applicationContext.xml'** en el directorio **'war/WEB-INF'**".

'springapp/war/WEB-INF/applicationContext.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.Añadir transaccion y una configuracion de pool de conexiones al contexto de la aplicacion

Siempre que persistas informacion en una base de datos es mejor usar transacciones para asegurarte que o todas o ninguna de tus actualizaciones son realizadas. Asi evitas tener la mitad de tus actualizaciones persistidas mientras la otra mitad ha fallado.

Spring ofrece un extenso margen de opciones para configurar mantenimiento de transacciones. El manual de referencia cubre este tema en profundidad. Aqui haremos uso de esta caracteristica usando AOP (Aspect Oriented Programming - Programacion Orientada a Aspectos) en la forma de un advice (consejo) de transaccion y un pointcut (punto de corte) AspectJ para definir donde deben ser aplicadas las transacciones. Si estas interesado en como funciona este mecanismo mas profundamente, echale un vistazo al manual de referencia. Vamos a usar el nuevo soporte para nombres de espacio introducido en Spring 2.0. Los nombres de espacio "aop" y "tx" hacen las entradas de configuracion mucho mas concisas comparadas que el sistema tradicional, el cual usa entradas de tipo "<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>
```

El pointcut es aplicado a cualquier metodo que invoques en la interface ProductManager. El advice es un advice de transaccion, y es aplicado a metodos cuyo nombre comience con 'save'. Son aplicados los atributos de configuracion por defecto (REQUIRED) puesto que ningun otro atributo ha sido especificado. El advice tambien es aplicado a transacciones "read-only" (de solo lectura) en cualquier otro metodo que sea alcanzado mediante el pointcut.

Tambien necesitamos definir un DataSource donde configuramos los parametros de conexion a la base de datos, asi como un configurador de propiedades, el cual leera dichos parametros desde el archivo **'jdbc.properties'** que hemos creado en la parte 5. Este Datasource usara automaticamente una conexion tipo pool, en nuestro caso una conexion DBCP del proyecto Apache Jakarta.

```
    <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>
```

Para que todo esto funcione necesitamos copiar algunos archivos jar en el directorio **'WEB-INF/lib'**. Copia **aspectjweaver.jar** desde el directorio **'spring-framework-2.5/lib/aspectj'** y **commons-dbcp.jar** y **commons-pool.jar** desde el directorio **'spring-framework-2.5/lib/jakarta-commons'** al directorio **'springapp/war/WEB-INF/lib'**.

Aqui esta la version final de nuestro archivo **'applicationContext.xml'**:

'springapp/war/WEB-INF/applicationContext.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. Test final de la aplicacion completa

Ahora es el momento de ver si todas estas piezas funcionan juntas. Construye y despliega la aplicacion finalizada y recuerda tener la base de datos arrancada y funcionando. Esto es lo que deberias ver cuando apuntes tu navegador web a la aplicacion despues de ser recargada:

La aplicacion completa

Aparece exactamente como lo hacia antes. Sin embargo hemos añadido la persistencia en base de datos, por lo que si cierras la aplicacion tus incrementos de precio no se perderan. Ellos estaran todavia alli cuando vuelvas a cargar la aplicacion.

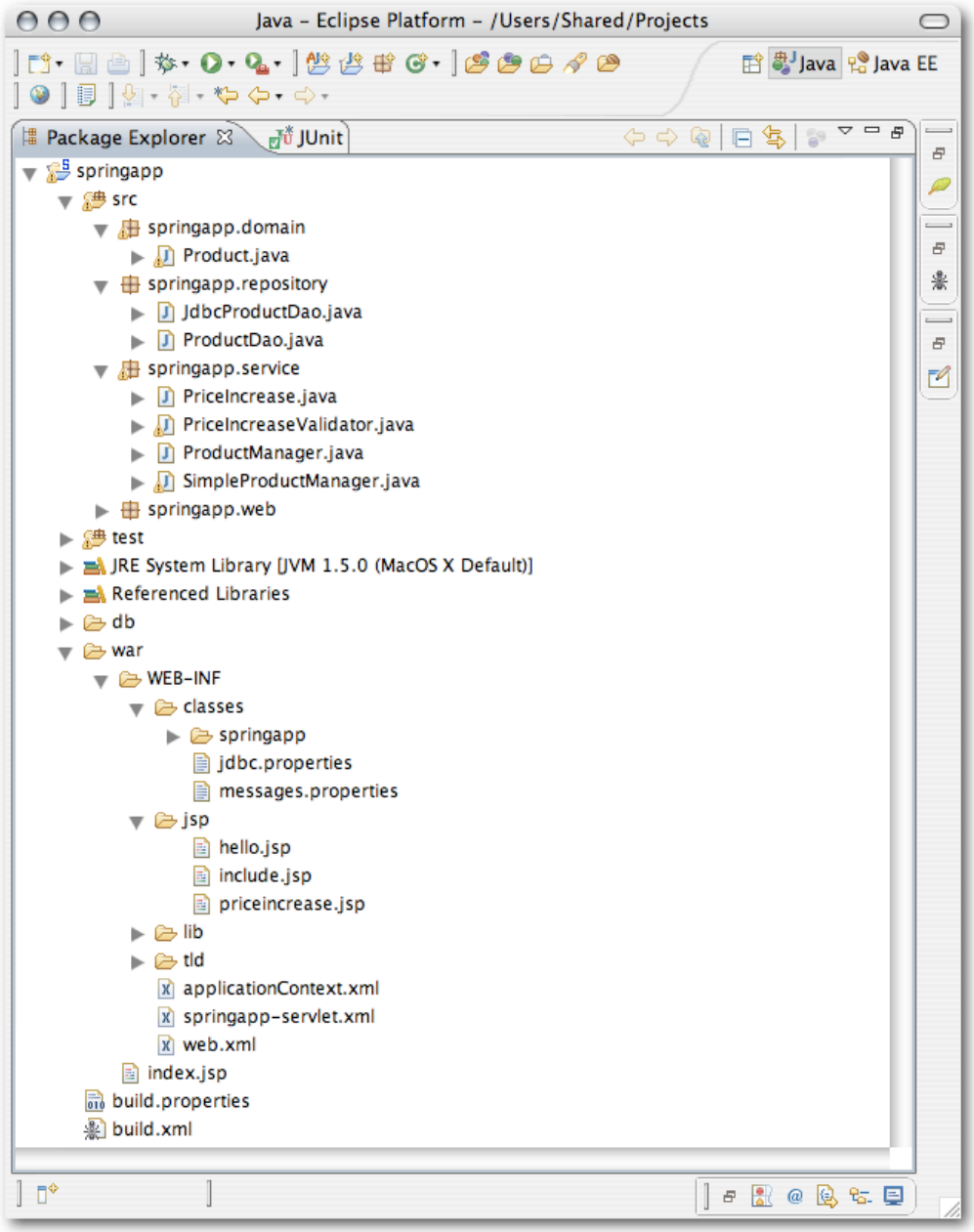
Un monton de trabajo para una aplicacion simple, pero nuestra meta nunca fue solamente escribir (¡y traducir!) la aplicacion. La meta fue mostrar como crear una aplicacion Spring MVC desde cero y ahora sabemos que las aplicaciones que tu construiras desde este momento seran mucho mas complejas. Deberas seguir los mismos pasos, y esperamos que hayas adquirido el conocimiento suficiente para hacerte mas facil comenzar a usar Spring.

6.6. Resumen

Hemos completado las tres capas de la aplicacion -- la capa web, la capa de servicio y la capa de persistencia. En esta ultima parte hemos reconfigurado la aplicacion.

- Primero hemos modificado la capa de servicio para usar la interface ProductDAO.
- Despues hemos tenido que arreglar algunos fallos en los tests de la capa de servicio y la capa web.
- A continuacion hemos introducido un nuevo applicationContext para separar la configuracion de la capa de servicio y de la capa de persistencia de la configuracion de la capa web.
- Hemos definido cierto mantenimiento de transacciones para la capa de servicio y configurado un pool de conexiones para las conexiones a la base de datos.
- Finalmente hemos construido la aplicacion y testeado que aun funciona.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.


Java - Eclipse Platform - /Users/Shared/Projects
Java
Java EE
Package Explorer
JUnit
springapp
src
springapp.domain
Product.java
springapp.repository
JdbcProductDao.java
ProductDao.java
springapp.service
PriceIncrease.java
PriceIncreaseValidator.java
ProductManager.java
SimpleProductManager.java
springapp.web
test
JRE System Library [JVM 1.5.0 (MacOS X Default)]
Referenced Libraries
db
war
WEB-INF
classes
springapp
jdbc.properties
messages.properties
jsp
hello.jsp
include.jsp
priceincrease.jsp
lib
tld
applicationContext.xml
springapp-servlet.xml
web.xml
index.jsp
build.properties
build.xml

Apendice A. Scripts Ant

Listado completo de `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="${build.dir}"/>
        <pathelement path="${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>
```

Listado completo de **build.properties**:

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.20
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=C:/Dev/apache-tomcat-6.0.20/lib

deploy.path=C:/Dev/apache-tomcat-6.0.20/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=
```

Inicio Home