

Práctica: Aplicación Spring básica

Objetivo: Crear una aplicación MVC con el entorno de desarrollo STS

Descripción

Este documento es una guía paso a paso sobre cómo desarrollar una aplicación web, partiendo de cero, usando Spring Framework.

Se asume que posees un conocimiento superficial de Spring, por lo que este tutorial te será útil si estas aprendiendo o investigando Spring. Durante el tiempo que trabajes a través del material presentado en el tutorial, podrás ver cómo encajan diversas partes de Spring Framework dentro de una aplicación web Spring MVC, como Inversión de Control (Inversion of Control - IoC), Programación Orientada a Aspectos (Aspect-Oriented Programming - AOP) y la integración con las diversas librerías de servicios (como JPA).

Spring provee diversas opciones para configurar tu aplicación. La forma más popular es usando archivos XML. Ésta es la forma más tradicional, soportada desde la primera versión de Spring. Con la introducción de Anotaciones en Java 5, ahora disponemos de una manera alternativa de configurar nuestras aplicaciones Spring. La nueva versión Spring 4.2.5 introduce un amplio soporte para configurar una aplicación web mediante anotaciones. Este documento usa el estilo más moderno de configuración mediante anotaciones.

Ten en cuenta que no se tratará ninguna información en profundidad en este tutorial, así como ningún tipo de teoría; hay multitud de libros disponibles que cubren Spring en profundidad; siempre que una nueva clase o característica sea usada en el tutorial, se mostrarán enlaces a la sección de documentación de Spring, donde la clase o característica es tratada en profundidad.

1. Contenido

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

- Inversión de Control (IoC)
- El framework Spring Web MVC
- Acceso a Datos mediante JPA
- Integración mediante tests

2. Software Requerido

Se requiere el siguiente software y su adecuada configuración en el entorno. Deberías sentirte razonablemente confortable usando las siguientes tecnologías.

- Java SDK 8+
- SpringSource Tool Suite 4.7.2
- Maven 3

El proyecto *SpringSource Tool Suite* (<https://spring.io/tools/sts/all>) proporciona un excelente entorno para el desarrollo de aplicaciones que utilicen *Spring Framework*. *SpringSource Tool Suite* integra sobre la plataforma *Eclipse* (<http://www.eclipse.org/>), entre otras cosas, el plugin de *Spring IDE* así como también da soporte al servidor Pivotal *tc Server* (Tomcat). Por supuesto, puedes usar cualquier variación de las versiones de software anteriores.

3. La aplicación que vamos a construir

La aplicación que vamos a construir desde cero a lo largo de este tutorial es un sistema de mantenimiento de inventario *básico*. Este sistema de mantenimiento de inventario está muy limitado en alcance y funcionalidad; abajo puedes ver un diagrama de casos de uso ilustrando los sencillos casos de uso que implementaremos. La razón por la que la aplicación es tan limitada es que puedas concentrarte en las características específicas de *Spring Web MVC* y *Spring*, y olvidar los detalles más sutiles del mantenimiento de inventario.

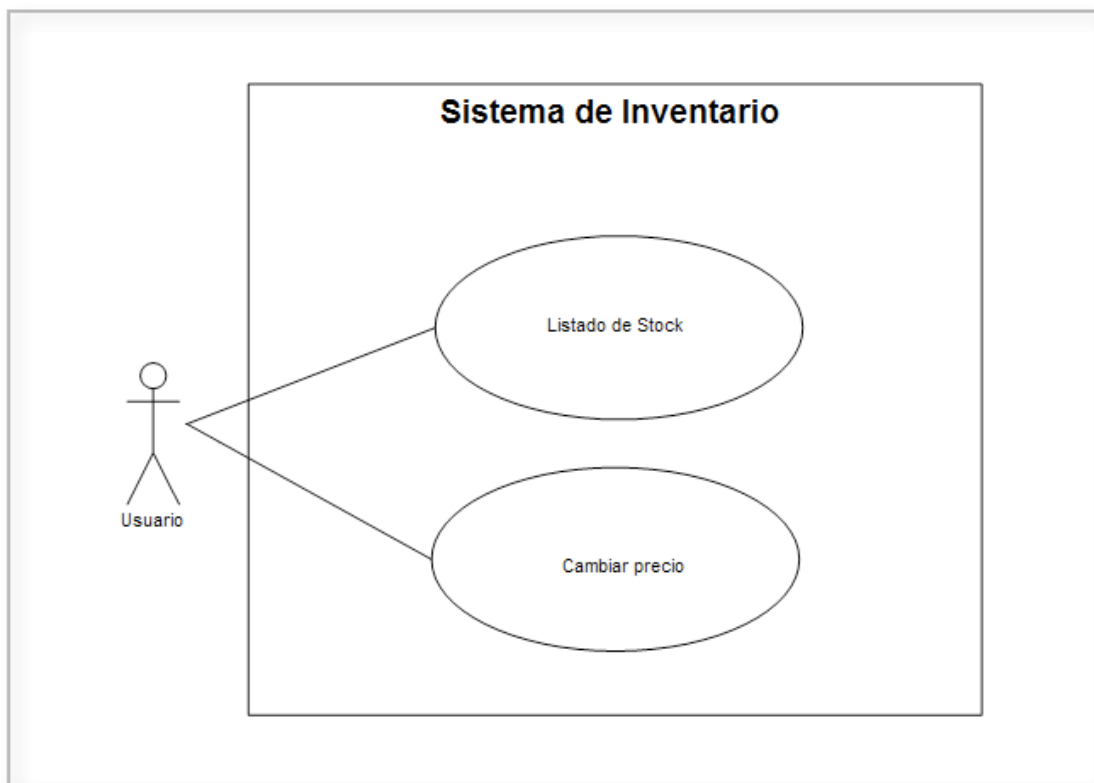


Diagrama de casos de uso de un sistema de mantenimiento de inventario

Comenzaremos configurando la estructura básica de los directorios para nuestra aplicación, descargando las librerías necesarias, etc. El primer paso nos

proporcionará una base sólida sobre la que desarrollar de forma adecuada las secciones 2, 3, y 4.

Una vez terminada la configuración básica, introduciremos Spring, comenzando con el framework Spring Web MVC. Usaremos Spring Web MVC para mostrar el stock inventariado, lo que implicará escribir algunas clases simples en Java y algunos JSP. Entonces introduciremos acceso de datos y persistencia en nuestra aplicación, usando para ello el soporte que ofrece Spring para JPA.

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

Capítulo 1. Aplicación Base y Configuración del Entorno

1.1. Crear la estructura de directorios del proyecto

Necesitamos crear un directorio donde alojar todos los archivos que vayamos creando, así que comenzaremos creando un proyecto en el Workspace llamado '**springapp**'. Utilizaremos Maven para facilitar la creación de la estructura de directorios del proyecto, así como la inclusión de las librerías necesarias.

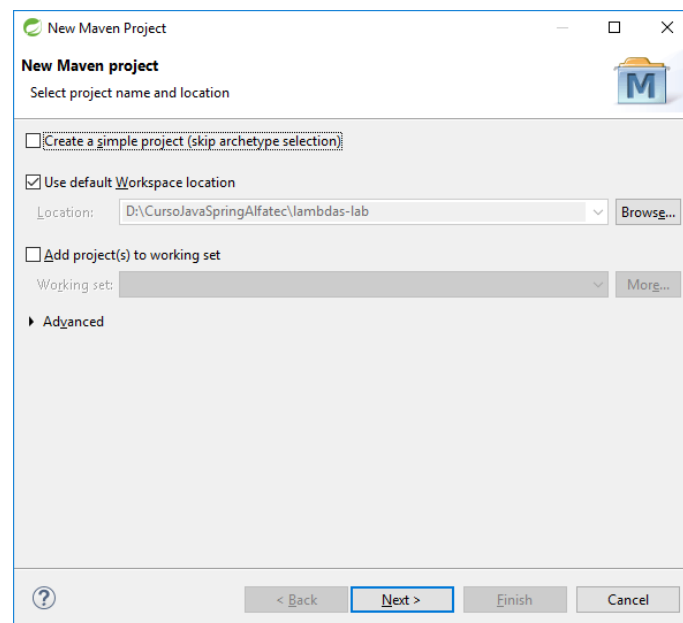


Figura 1. Creación del nuevo proyecto usando Maven.

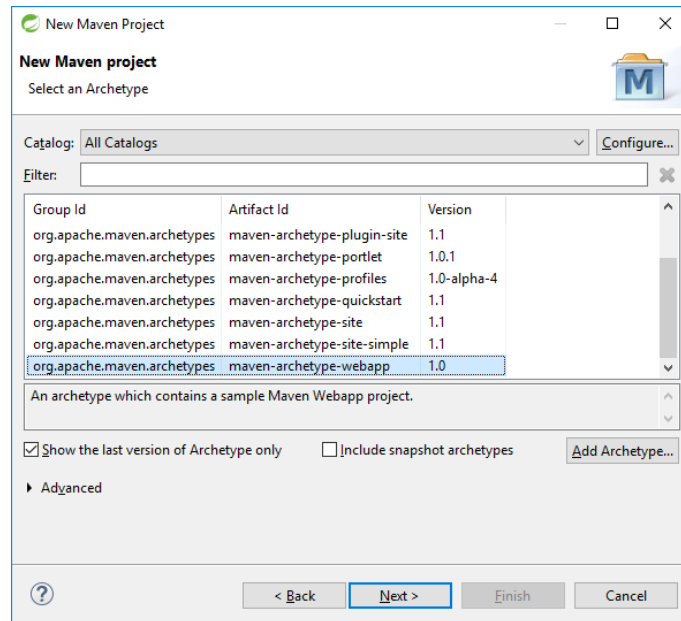


Figura 2. Selección del arquetipo del proyecto.

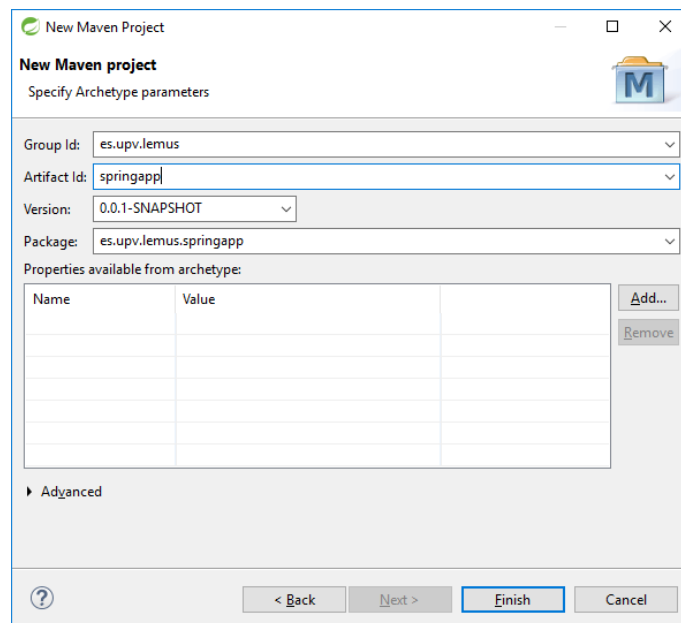


Figura 3. Definición del nombre del proyecto.

Aparecerá un warning en el proyecto causado por el uso de diferentes versiones de la JDK, ajustarlas conveniente configurando el **Build Path** del proyecto. En las últimas versiones de STS también aparece un error en el fichero JSP porque todavía no se ha incorporado la dependencia 'servlet-api'.

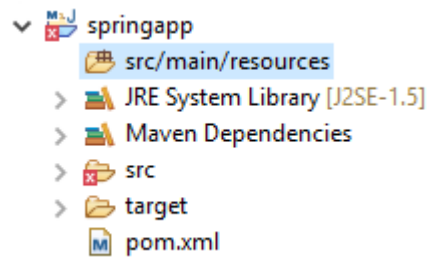
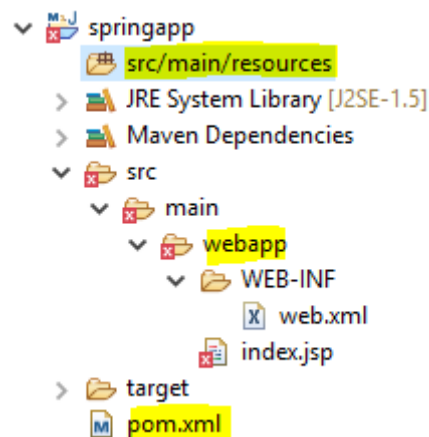


Figura 4. Estructura de directorios creada.

Se puede seguir adelante con el tutorial a pesar de este error, ya que lo solucionaremos en el [paso 1.5](#).

Una vez que el proyecto se ha creado, podemos identificar la siguiente estructura:

- El subdirectorio '**src/main/resources**' que contendrá todos los recursos utilizados por la aplicación.
- El subdirectorio '**src/main/webapp**' que alojará todos los archivos que no sean código fuente Java, como archivos JSP y de configuración.
- El directorio '**target**' donde se generará el archivo WAR que usaremos para almacenar y desplegar rápidamente nuestra aplicación.
- El fichero '**pom.xml**' que contiene las dependencias Maven.



A continuación puedes ver una captura de pantalla que muestra como quedaría la estructura de directorios si has seguido las instrucciones correctamente. (La imagen muestra dicha estructura desde el SpringSource Tool Suite (STS)).

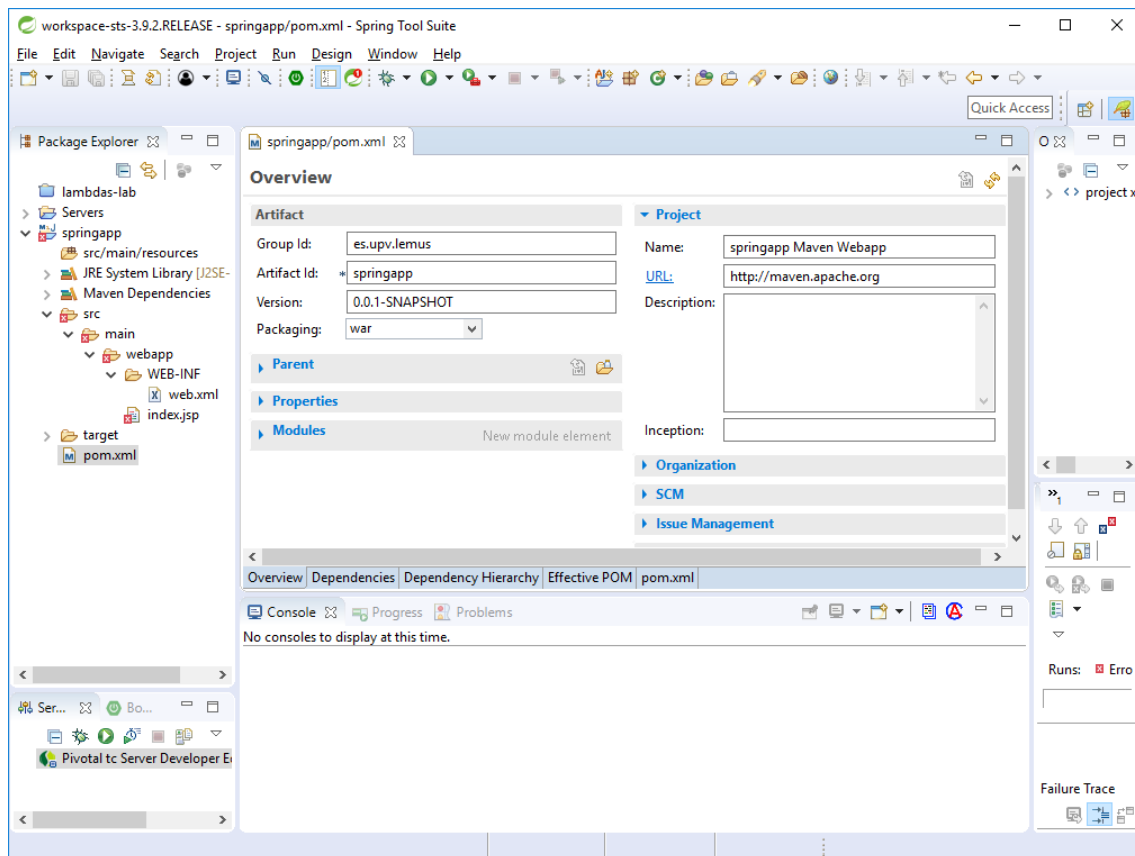


Figura 5. Estructura de directorios en el STS.

1.2. Crear 'index.jsp'

Puesto que estamos creando una aplicación web, Maven ya ha creado un archivo JSP muy simple llamado **'index.jsp'** en el directorio **'src/main/webapp'**.

El archivo **'index.jsp'** es el punto de entrada a nuestra aplicación. Para familiarizarnos con el entorno, podemos cambiarlo por el que se muestra a continuación:

```
<html>
  <body>
    <title>Aplicación Básica</title>
    <h2>Este es el inicio</h2>
  </body>
</html>
```

Listado 1. Fichero springapp/src/main/webapp/WEB-INF/web.xml'

Asimismo, Maven también ha creado un archivo llamado **'web.xml'** dentro del directorio **'src/main/webapp/WEB-INF'** con la configuración básica para ejecutar la aplicación por primera vez. Proponemos modificarlo por el que se muestra a continuación para utilizar una especificación más moderna de JavaEE.

1.3. Desplegar la aplicación en el servidor

Para compilar, construir y desplegar la aplicación automáticamente sólo es necesario seleccionar 'Run as > Run on Server' sobre el menú contextual que aparece cuando se pincha el botón derecho sobre el nombre del proyecto. A continuación, debemos seleccionar el servidor desde el cuadro de diálogo que ofrece los servidores dados de alta en el entorno.

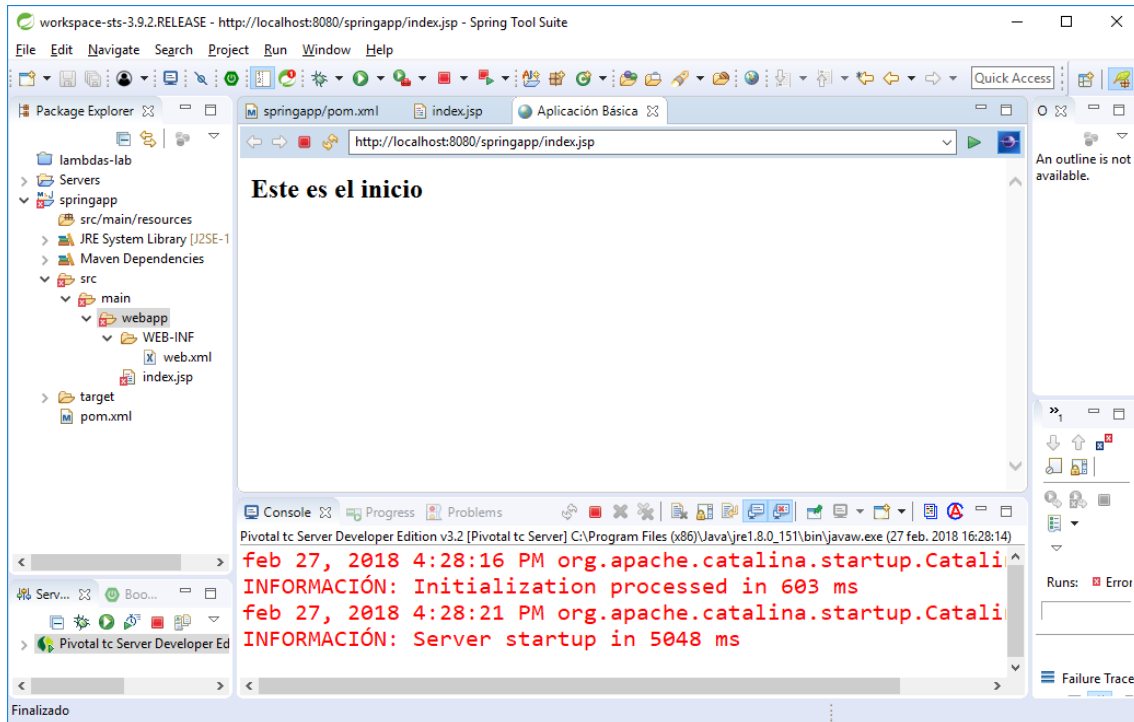


Figura 6. página de inicio de la aplicación.

Se recomienda detener el servidor para evitar la recarga automática mientras se sigue desarrollando el proyecto.

1.5. Descargar Spring Framework

Utilizaremos Maven para gestionar las dependencias del proyecto con Spring Framework así como para descargar otras librerías adicionales necesarias. Se puede consultar [MVNrepository](http://mvnrepository.com) para obtener los datos concretos de las dependencias que incluiremos en el fichero 'pom.xml'.

En este fichero, hemos introducido una propiedad llamada 'org.springframework.version' y le hemos dado el valor '4.3.14.RELEASE'. Este valor corresponde con la última versión disponible de Spring Framework en el momento en el que ha escrito el tutorial.

Para consultar las versiones disponibles, se puede hacer una búsqueda en Sonatype de la dependencia (artifact) 'spring-core'. La introducción de propiedades con el valor de la versión de cada dependencia facilitará la actualización del proyecto a nuevas versiones.

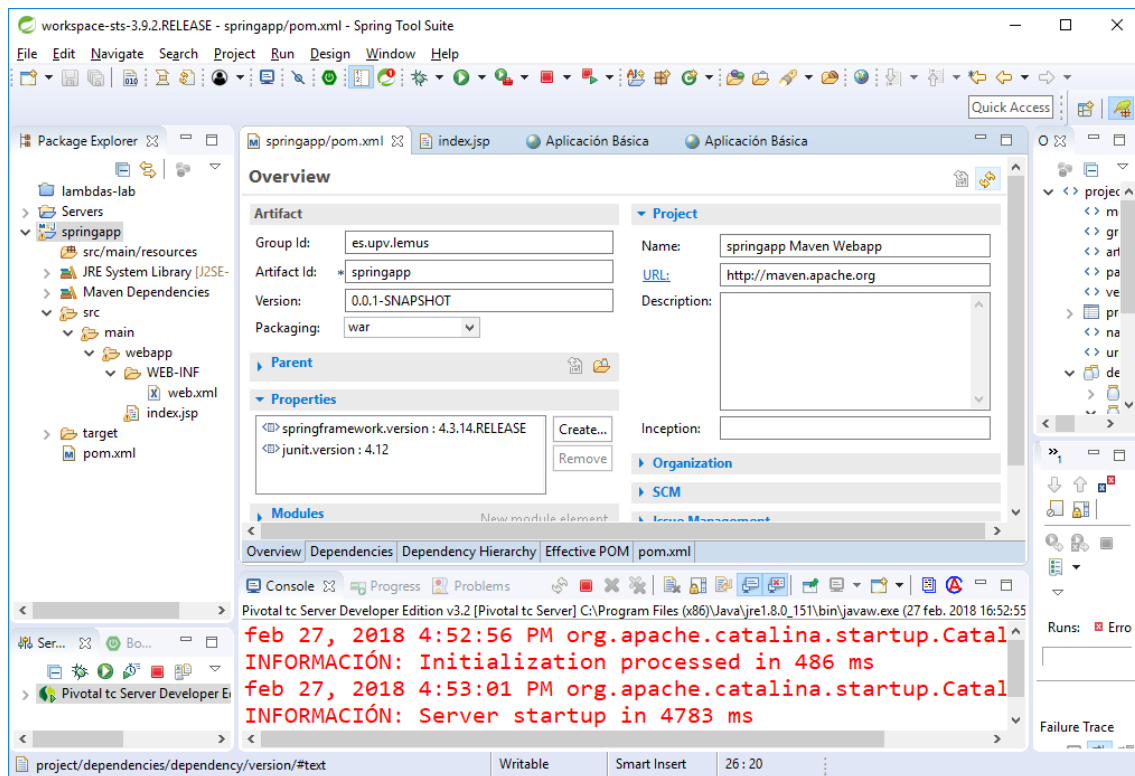


Figura 7. Propiedades del fichero pom.

En este proyecto se han creado las siguientes dependencias 'Dependencies' del fichero '**pom.xml**'.

Las librerías serán descargadas y añadidas automáticamente al proyecto en el momento en el que guardemos el fichero '**pom.xml**'

Group Id	Artifact Id	Version	Scope
junit	junit	4.12	Test
org.springframework	spring-core	\${springframework.version}	Compile
org.springframework	spring-webmvc	\${springframework.version}	Compile
javax.servlet	servlet-api	2.5	Provided

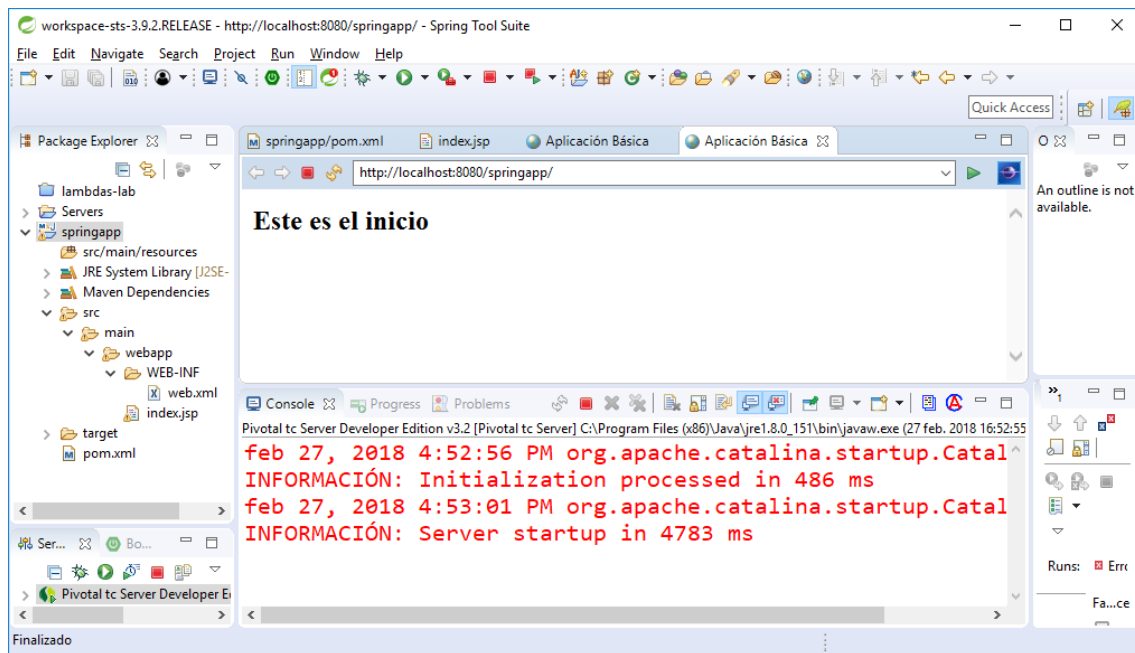


Figura 8. Ejecución de la aplicación con las dependencias.

Nótese como la dependencia 'junit' (incluida por defecto en el fichero 'pom.xml') ha sido actualizada a la versión '4.12'. Por otra parte, la dependencia 'servlet-api' ha sido marcada como 'Provided', ya que será proporcionada por el servidor sobre el que se despliegue la aplicación.

1.6. Modificar 'web.xml' en el directorio 'src/main/webapp/WEB-INF'

Sitúate en el directorio 'src/main/webapp/WEB-INF'. Modifica el archivo 'web.xml' del que hemos hablado anteriormente. Vamos a definir un DispatcherServlet (también llamado 'Controlador Frontal' (Crupi et al)).

Su misión será controlar hacia dónde serán enrutadas todas nuestras solicitudes basándose en información que introduciremos posteriormente. La definición del servlet tendrá como acompañante una entrada <servlet-mapping/> que mapeará las URL que queremos que apunten a nuestro servlet. Hemos decidido permitir que cualquier URL con una extensión de tipo '.htm' sea enrutada hacia el servlet 'springapp' (DispatcherServlet).

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>springapp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/app-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>springapp</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>
</web-app>
```

```

</servlet>

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

```

Listado 2. Fichero springapp/src/main/webapp/WEB-INF/web.xml'

A continuación, creamos el subdirectorio '**src/main/webapp/WEB-INF/spring**' y dentro el archivo llamado '**app-config.xml**'. Este archivo contendrá las definiciones de beans (POJO's) usados por el `DispatcherServlet`. Es decir, este archivo es el `WebApplicationContext` donde situaremos todos los componentes. Por tanto, utilizaremos el asistente para la creación de ficheros de tipo 'Spring Bean Configuración File'. *(Si no se utiliza STS y no se dispone del plugin Spring IDE de Eclipse, basta crear un fichero xml como el que se mostrará a continuación).*

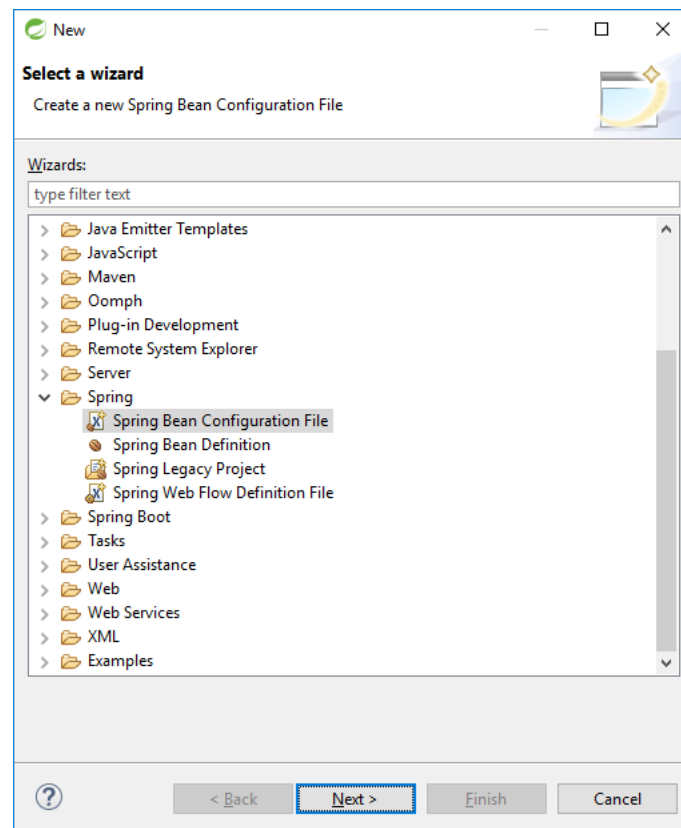


Figura 9. Creación del fichero de Configuración de Spring.

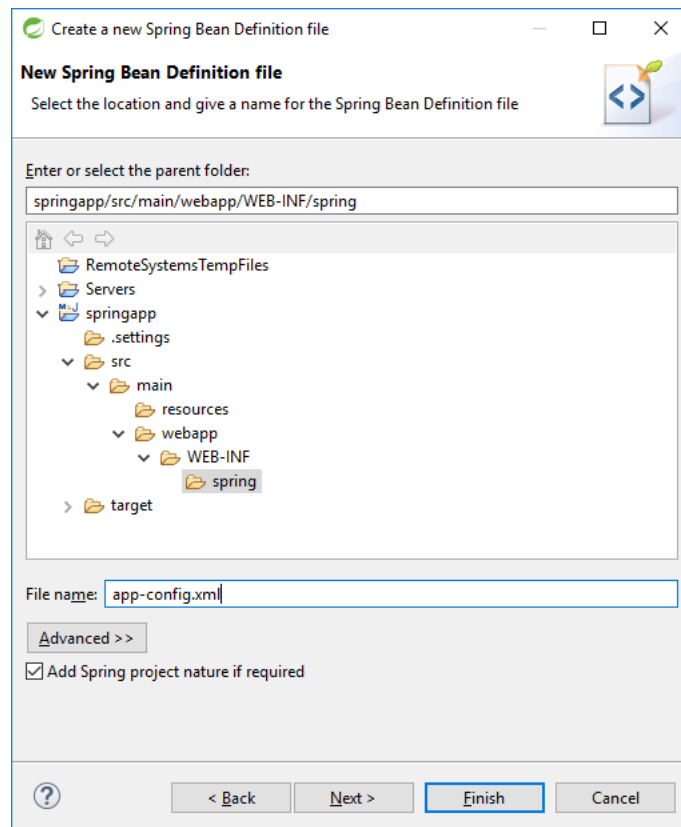


Figura 10. Creación del fichero de Configuración de Spring (continuación).

Tras haber introducido el nombre del fichero, es necesario introducir los namespaces que se utilizarán. Seleccionamos los namespaces 'beans', 'context' y 'mvc' en las versiones que corresponden con la versión '4.0' de Spring Framework.

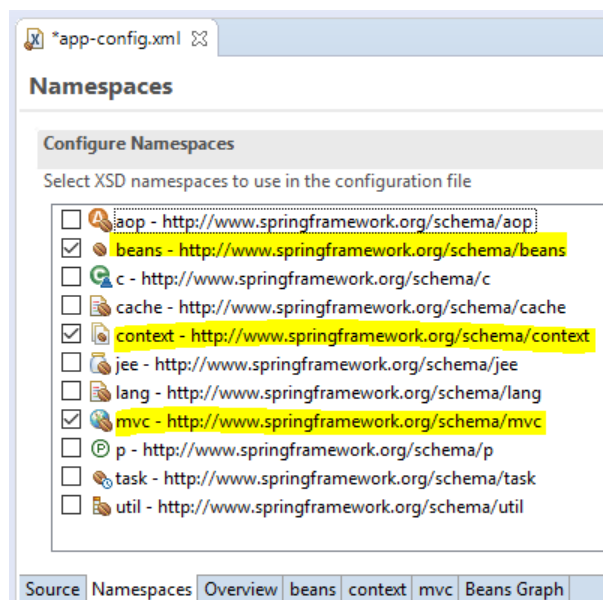


Figura 11. Selección de los namespaces del fichero 'app-config.xml'.

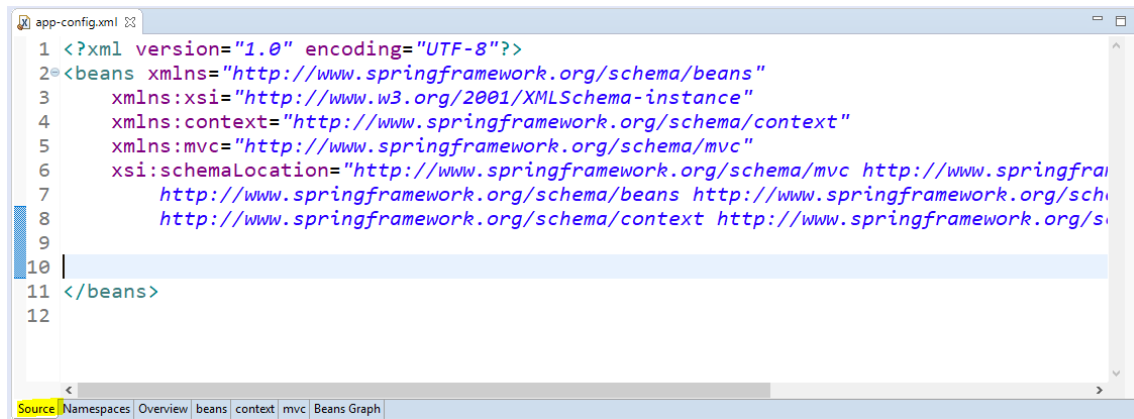


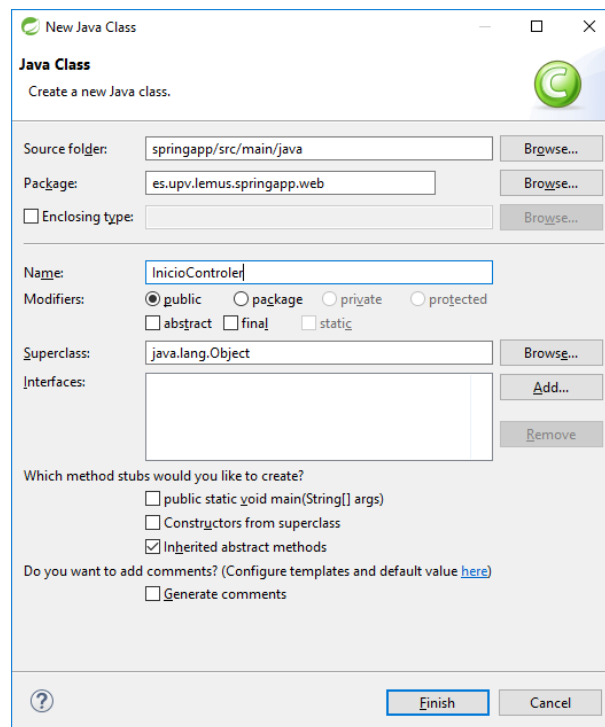
Figura 12. Espacios de nombres creados en el fichero 'app-config.xml'.

Por último, activamos la detección automática de componentes a través del uso de anotaciones. El fichero 'app-config.xml', por tanto, deberá tener el siguiente aspecto.

1.7. Crear el Controlador

Primero debemos crear la carpeta springapp/src/main/java

A continuación, vamos a crear una clase anotada con la etiqueta @Controller (a la que llamaremos InicioControlador.java) y que estará definida dentro del paquete 'es.upv.lemus.springapp.web' del directorio 'springapp/src/main/java'



El contenido del fichero controlador es:

```

1 package es.upv.lemus.springapp.web;
2
3 import java.io.IOException;
4
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8
9 import org.springframework.stereotype.Controller;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import org.springframework.web.servlet.ModelAndView;
12
13 @Controller
14 public class InicioControlador {
15
16     @RequestMapping(value="/inicio.htm")
17     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
18         throws ServletException, IOException {
19         return new ModelAndView("inicio.jsp");
20     }
21 }
22

```

Figure 1. Contenido del fichero InicioControlador.java.

Contenido de la página src/main/webapp/inicio.jsp

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
7 <meta name="author" content="Lenin Lemus">
8 <meta name="viewport" content="width=device-width, initial-scale=1.0">
9 <title>Curso Spring</title>
10 </head>
11 <body>
12     <h1>Curso Spring 2018</h1>
13     <p>2018 &copy;Lenin Lemus</p>
14 </body>
15 </html>

```

Figure 2. Contenido del fichero src/main/webapp/inicio.jsp.

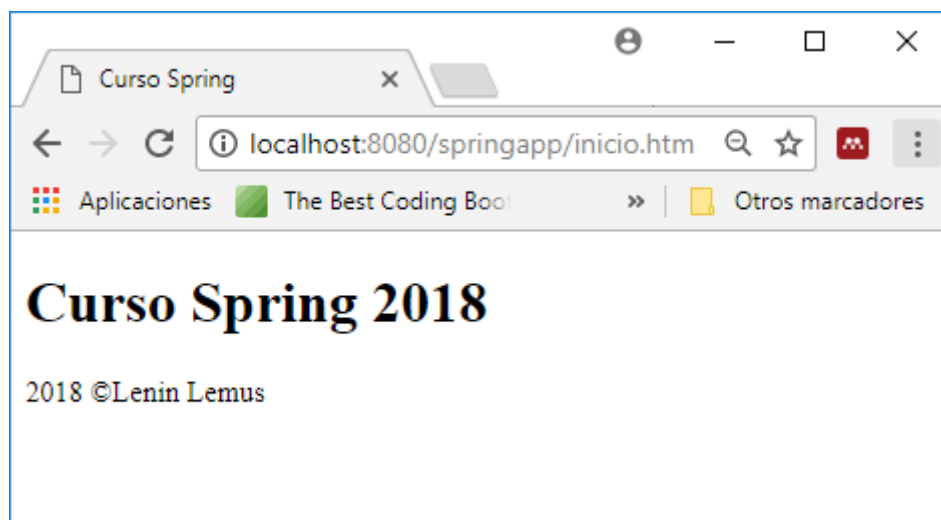


Figure 3. Ejecución de la aplicación.

1.11. Resumen

Echemos un vistazo rápido a las partes de nuestra aplicación que hemos creado hasta ahora.

- Una página de inicio, '**index.jsp**', la página de bienvenida de nuestra aplicación. Fue usada para comprobar que nuestra configuración era correcta. Más tarde la cambiaremos para proveer un enlace a nuestra aplicación.
- Un controlador frontal, `DispatcherServlet`, con el correspondiente archivo de configuración '**app-config.xml**'.
- Un controlador de página, `InicioControlador`, con funcionalidad limitada – simplemente devuelve un objeto `ModelAndView`. Actualmente tenemos un modelo vacío, más tarde proveeremos un modelo completo.
- Una vista, '**inicio.jsp**', que de nuevo es extremadamente sencilla. Las buenas noticias son que el conjunto de la aplicación funciona y que estamos listos para añadir más funcionalidad.

A continuación, puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.

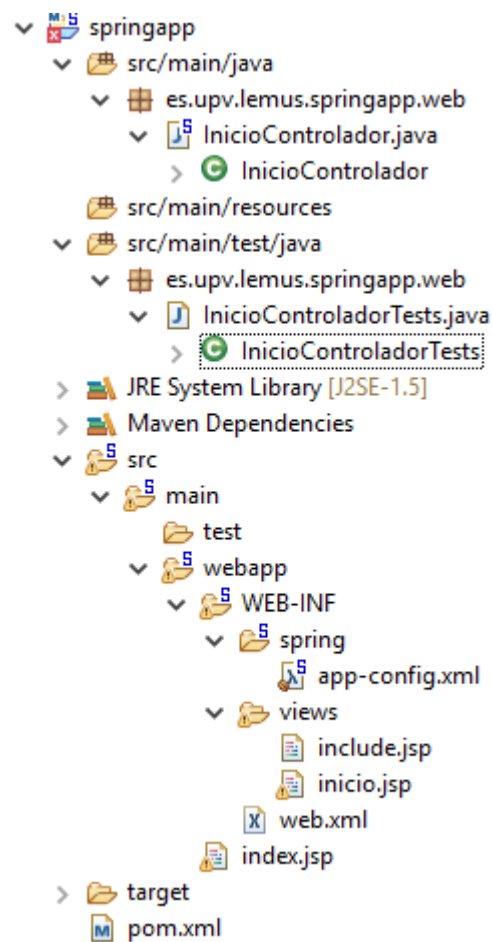


Figure 4. Estructura de ficheros del proyecto.

Capítulo 2. Desarrollando y Configurando la Vista y el Controlador

Ésta es la Parte 2 del tutorial paso a paso sobre cómo desarrollar una aplicación web desde cero usando Spring Framework. En la [Parte 1](#) hemos configurado el entorno y montado una aplicación básica que ahora vamos a desarrollar.

2.1. Configurar JSTL y añadir un archivo de cabecera JSP

Vamos a usar la Librería Estándar JSP (JSP Standard Tag Library - JSTL), así que comencemos definiendo la dependencia que necesitamos en el fichero `pom.xml`

Group Id	Artifact Id	Version	Scope
javax.servlet.jsp.jstl	jstl-api	1.2	Compile
org.apache.taglibs	taglibs-standard-impl	1.2.5	Compile

Vamos a crear un archivo de 'cabecera' que será embebido en todas las paginas JSP que escribamos después. Así estaremos seguros de que las mismas definiciones son incluidas en todos nuestros JSP al insertar el archivo de cabecera. También vamos a poner todos nuestros archivos JSP en un directorio llamado **'views'** bajo el directorio **'src/main/webapp/WEB-INF'**. Esto asegurará que nuestras vistas sólo puedan ser accedidas a través del controlador, por lo que no será posible acceder a estas páginas a través de una dirección URL. Esta estrategia podría no funcionar en algunos servidores de aplicaciones; si ése es tu caso, mueve el directorio **'views'** un nivel hacia arriba y usa **'src/main/webapp/views'** como el directorio de JSP en todos los ejemplos que encontrarás a lo largo del tutorial, en lugar de **'src/main/webapp/WEB-INF/views'**.

Primero creamos un archivo de cabecera para incluir en todos los archivos JSP que crearemos con posterioridad. Llama al fichero **'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" %>
```

Figure 5. Contenido del fichero `src/main/webapp/inicio.jsp`.

Ahora podemos actualizar **'index.jsp'** para que incluya este archivo, y puesto que estamos usando JSTL, podemos usar la etiqueta `<c:redirect/>` para redireccionar hacia nuestro controlador frontal: `InicioControlador`. Esto significa que todas nuestras solicitudes a **'index.jsp'** se resolverán a través de dicho controlador. Elimina los contenidos actuales de **'index.jsp'** y reemplázalos con los siguientes:

Crea el directorio

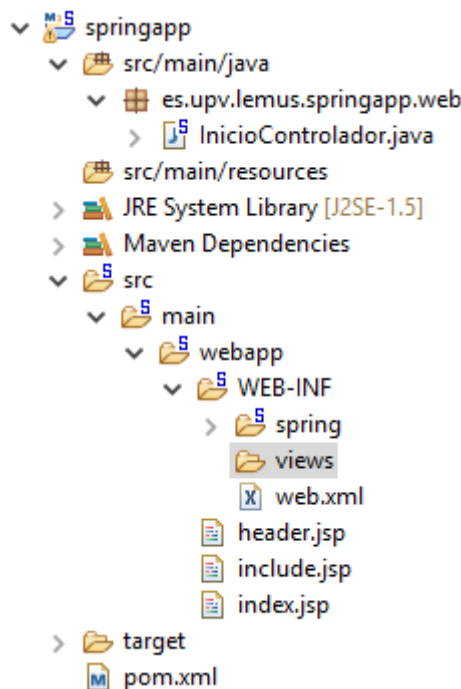


Figure 6. Estado de los ficheros del proyecto.

'src/main/webapp/WEB-INF/views'. Añade la misma directiva include que hemos añadido en 'index.jsp' a 'inicio.jsp'. Vamos a añadir también la fecha y hora actual, que serán leídas desde el modelo que pasaremos a la vista, y que mostraremos usando la etiqueta JSTL `<c:out/>`.

2.2. Mejorar el controlador

Antes de actualizar la localización del JSP en nuestro controlador, actualicemos nuestra unidad de test. Sabemos que necesitamos actualizar la referencia a la vista con su nueva localización, '**WEB-INF/views/inicio.jsp**'. También sabemos que debería haber un objeto en el modelo mapeado a la clave "now".

```
InicioControladorTests.java
1 package es.upv.lemus.springapp.web;
2
3 import static org.junit.Assert.assertEquals;
4
10
11 public class InicioControladorTests {
12     transient private Logger logger = LoggerFactory.getLogger(InicioControladorTests.class);
13     @Test
14     public void testHandleRequestView() throws Exception{
15         InicioControlador controller = new InicioControlador();
16         ModelAndView modelAndView = controller.handleRequest(null, null);
17         assertEquals("/WEB-INF/views/inicio.jsp", modelAndView.getViewName());
18         assertNotNull(modelAndView.getModel());
19         String nowValue = (String) modelAndView.getModel().get("now");
20         logger.info("Valor de now:" + nowValue);
21         assertNotNull(nowValue);
22     }
23 }
24
```

Figure 7. Test Unitario del controlador.

Si ejecutamos el test unitario fallará, porque falta modificar el controlador. A continuación, se muestra como modificar el controlador

```
1 package es.upv.lemus.springapp.web;
2
3 import java.io.IOException;
4 import java.util.Date;
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8
9 import org.slf4j.Logger;
10 import org.slf4j.LoggerFactory;
11 import org.springframework.stereotype.Controller;
12 import org.springframework.web.bind.annotation.RequestMapping;
13 import org.springframework.web.servlet.ModelAndView;
14
15 @Controller
16 public class InicioControlador {
17     transient private Logger logger = LoggerFactory.getLogger(InicioControlador.class);
18     @RequestMapping(value="/inicio.htm")
19     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
20         throws ServletException, IOException {
21         String now = (new Date()).toString();
22         logger.info("La hora actual es: " + now);
23         return new ModelAndView("/WEB-INF/views/inicio.jsp", "now", now);
24     }
25 }
```

Figure 8. Modificaciones al controlador.

La ejecución del programa debe ser la siguiente:

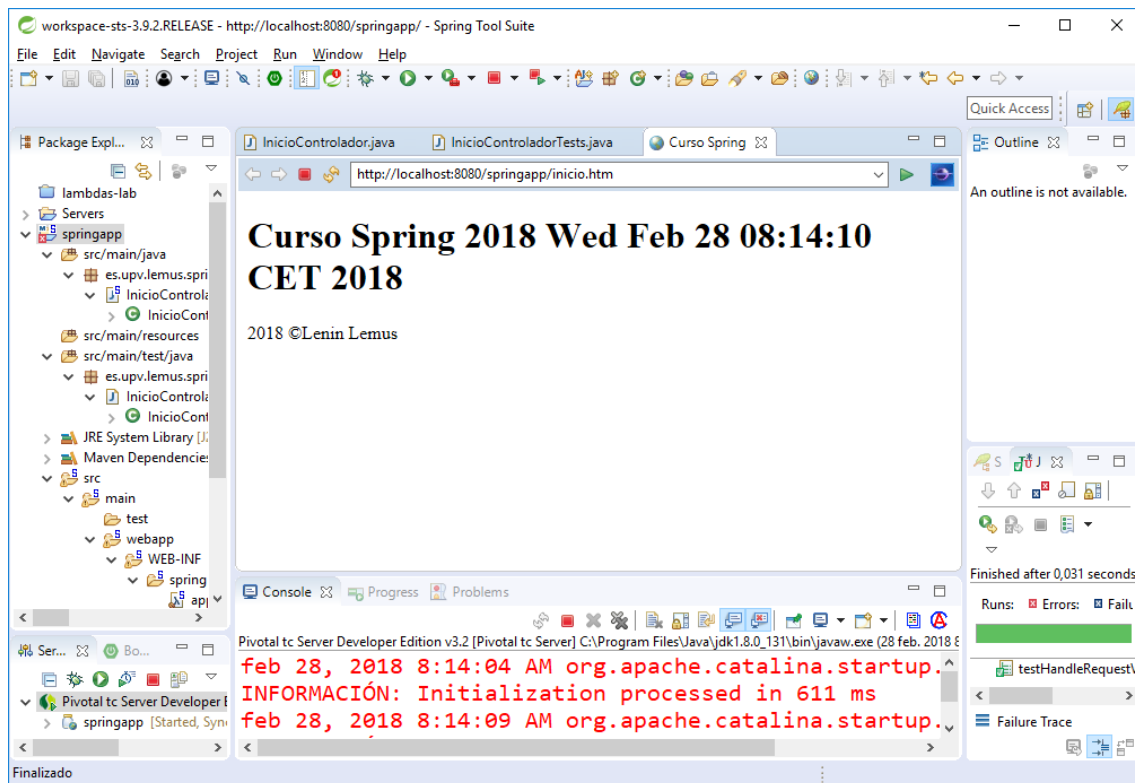
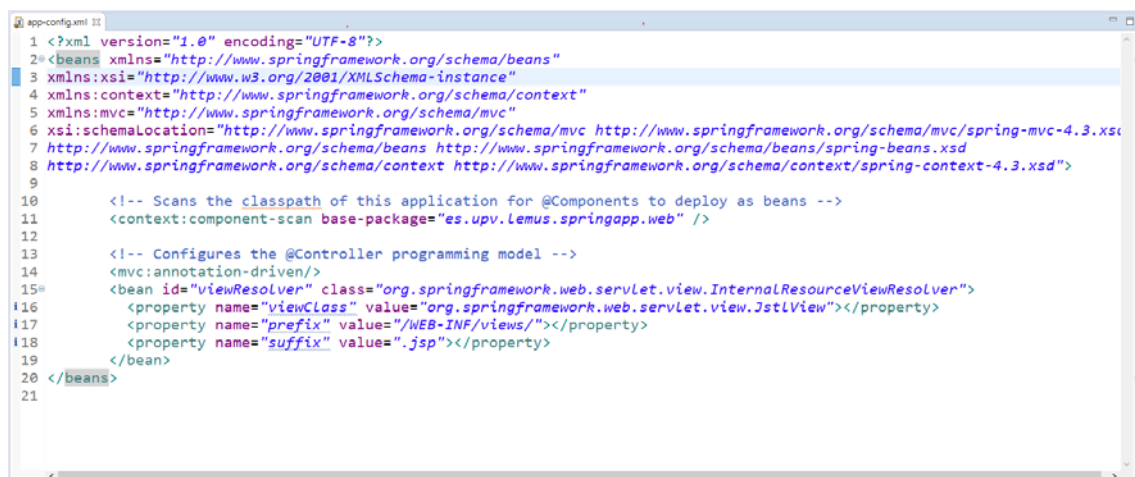


Figure 9. Ejecución de la aplicación básica.

2.3. Separar la vista del controlador

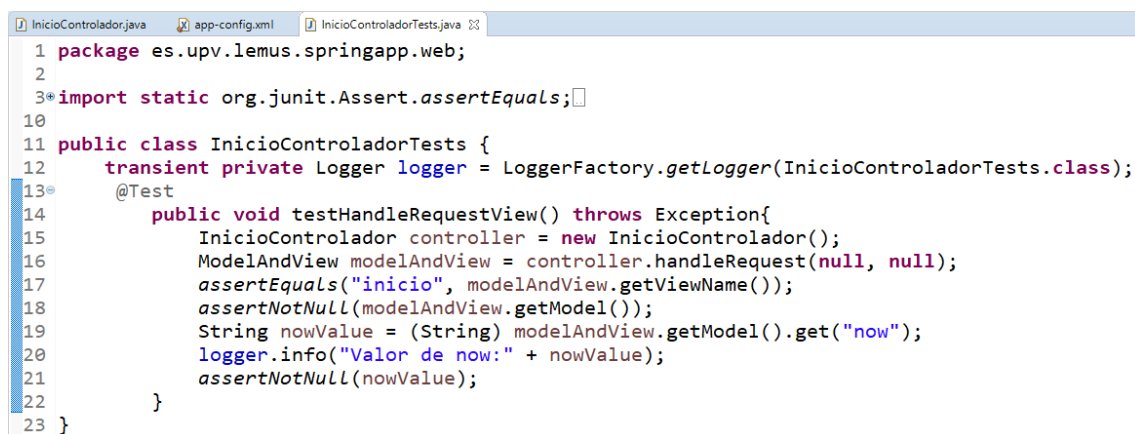
Ahora el controlador especifica la ruta completa a la vista, lo cual crea una dependencia innecesaria entre el controlador y la vista. Idealmente queremos referirnos a la vista usando un nombre lógico, permitiéndonos intercambiar la vista sin tener que cambiar el controlador. Puedes crear este mapeo en un archivo de propiedades si estas usando `ResourceBundleViewResolver` y `SimpleUrlHandlerMapping`. Otra opción para el mapeo básico entre una vista y una localización, consiste en simplemente configurar un prefijo y sufijo en `InternalResourceViewResolver`. Esta solución es la que vamos a implantar ahora, por lo que modificamos `'app-config.xml'` y declaramos una entrada `'viewResolver'`. Eligiendo `JstlView` tendremos la oportunidad de usar JSTL en combinación con paquetes de mensajes de idioma, los cuales nos ofrecerán soporte para la internacionalización de la aplicación.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
5       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">
6   <!-- Scans the classpath of this application for @Components to deploy as beans -->
7   <context:component-scan base-package="es.upv.lemus.springapp.web" />
8   <!-- Configures the @Controller programming model -->
9   <mvc:annotation-driven/>
10  <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
11    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"></property>
12    <property name="prefix" value="/WEB-INF/views/"></property>
13    <property name="suffix" value=".jsp"></property>
14  </bean>
15 </beans>
```

Figure 10. Agregamos un ViewResolver.

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



```
1 package es.upv.lemus.springapp.web;
2
3 import static org.junit.Assert.assertEquals;
4
5
6 public class InicioControladorTests {
7     @Test
8     public void testHandleRequestView() throws Exception{
9         InicioControlador controller = new InicioControlador();
10        ModelAndView modelAndView = controller.handleRequest(null, null);
11        assertEquals("inicio", modelAndView.getViewName());
12        assertNotNull(modelAndView.getModel());
13        String nowValue = (String) modelAndView.getModel().get("now");
14        logger.info("Valor de now:" + nowValue);
15        assertNotNull(nowValue);
16    }
17 }
```

Figure 11. Modificar el Test Unitario.

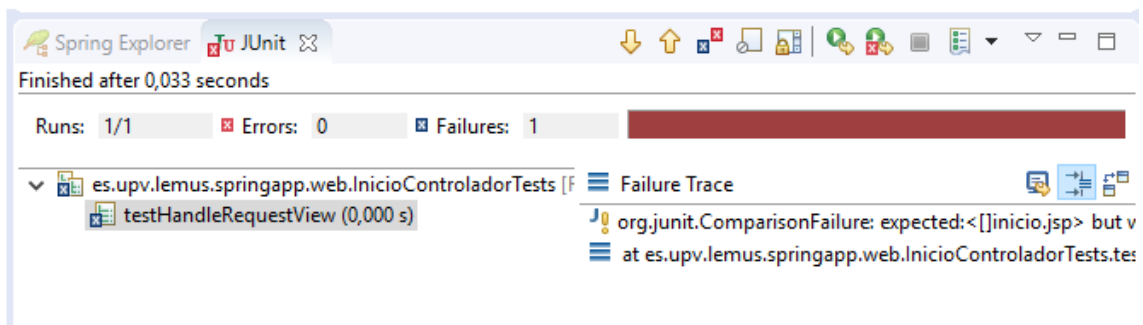


Figure 12. Fallo del test Unitario.

Ahora eliminamos el prefijo y sufijo del nombre de la vista en el controlador, dejando que el controlador se refiera a la vista por su nombre lógico "inicio".

```

1 package es.upv.lemus.springapp.web;
2
3 import java.io.IOException;
4
5 @Controller
6 public class InicioControlador {
7     transient private Logger logger = LoggerFactory.getLogger(InicioControlador.class);
8     @RequestMapping(value="/inicio.htm")
9     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
10         throws ServletException, IOException {
11         String now = (new Date()).toString();
12         logger.info("La hora actual es: " + now);
13         return new ModelAndView("inicio.jsp", "now", now);
14     }
15 }
16

```

Figure 13. Modificaciones al controlador.

Ahora la aplicación debe funcionar como se esperaba.

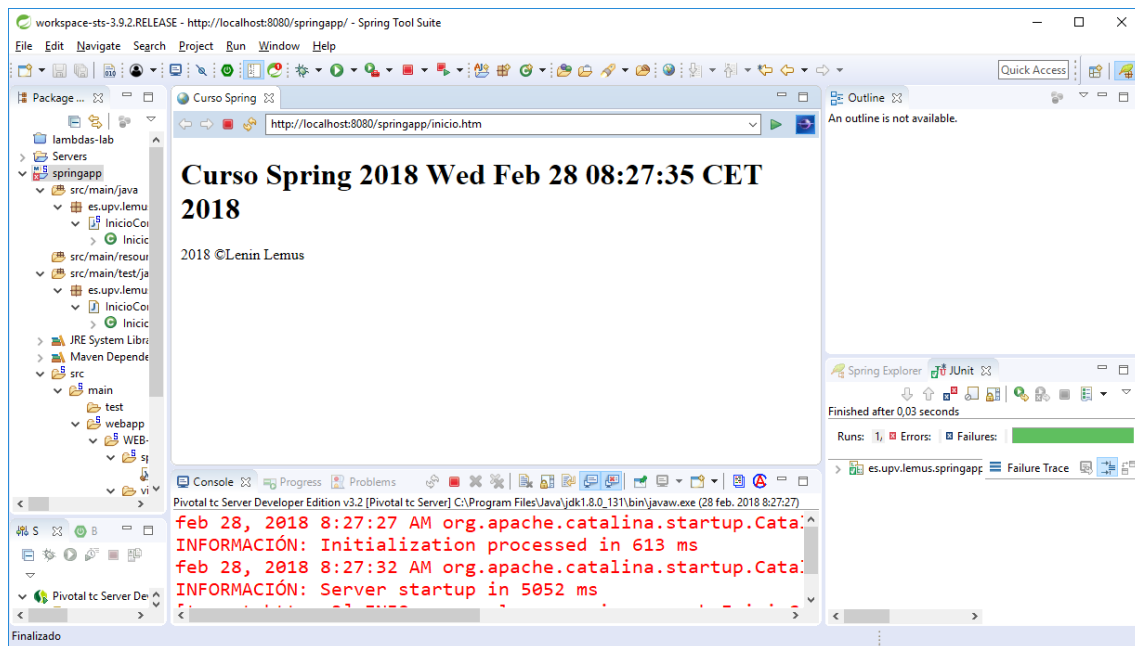


Figure 14. Aspecto de la aplicación.

2.4. Resumen

Analicemos 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 aplicación que hemos cambiado en la Parte 2.

- `InicioControladorTests` ha sido actualizado repetidamente para hacer al controlador referirse al nombre lógico de la vista en lugar de a su localización y nombre completo.

El controlador de página, `InicioControlador`, ahora hace referencia a la vista por su nombre lógico mediante el uso del '`InternalResourceViewResolver`' definido en '**app-config.xml**'.

A continuación, puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.

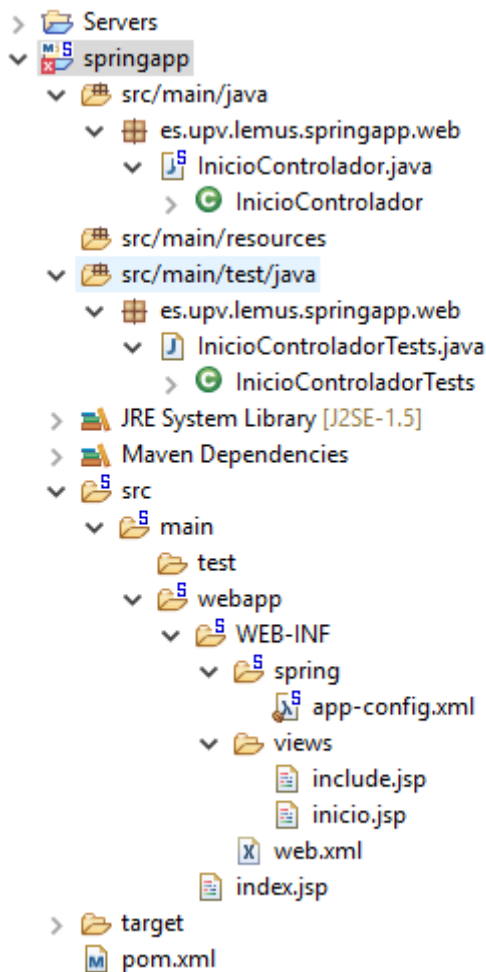


Figure 15. Estructura de ficheros.

Capítulo 3. Desarrollando la Lógica de Negocio

Ésta es la Parte 3 del tutorial paso a paso para desarrollar una aplicación Spring MVC. En esta sección, adoptaremos un acercamiento pragmático al Test-Driven Development (TDD o Desarrollo Conducido por Tests) para crear los objetos de dominio e implementar la lógica de negocio para nuestro sistema de mantenimiento de inventario. Esto significa que "escribiremos un poco de código, lo testaremos, escribiremos un poco más de código, lo volveremos a testear..." En la Parte 1 hemos configurado el entorno y montado la aplicación básica. En la Parte 2 hemos refinado la aplicación desacoplando la vista del controlador.

Spring permite hacer las cosas simples fáciles y las difíciles 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 implementación de interfaces). Los POJOs son objetos normales Java que están libres de dichas obligaciones, haciendo la programación orientada a objetos posible de nuevo. Cuando trabajas con

Spring, los objetos de dominio y los servicios que implementes serán POJOs. De hecho, casi todo lo que implementes debería ser un POJO. Si no es así, deberías preguntarte a ti mismo por qué no ocurre esto. En esta sección, 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. Supongamos en este tutorial que el negocio solicita la capacidad de incrementar precios sobre todos los productos. Cualquier decremento será hecho sobre cada producto en concreto, pero esta característica está fuera de la funcionalidad de nuestra aplicación. Las reglas de validación para incrementar precios son:

- El incremento máximo está limitado al 50%.
- El incremento mínimo debe ser mayor del 0%.

A continuación, puedes ver el diagrama de clases de nuestro sistema de mantenimiento de inventario.

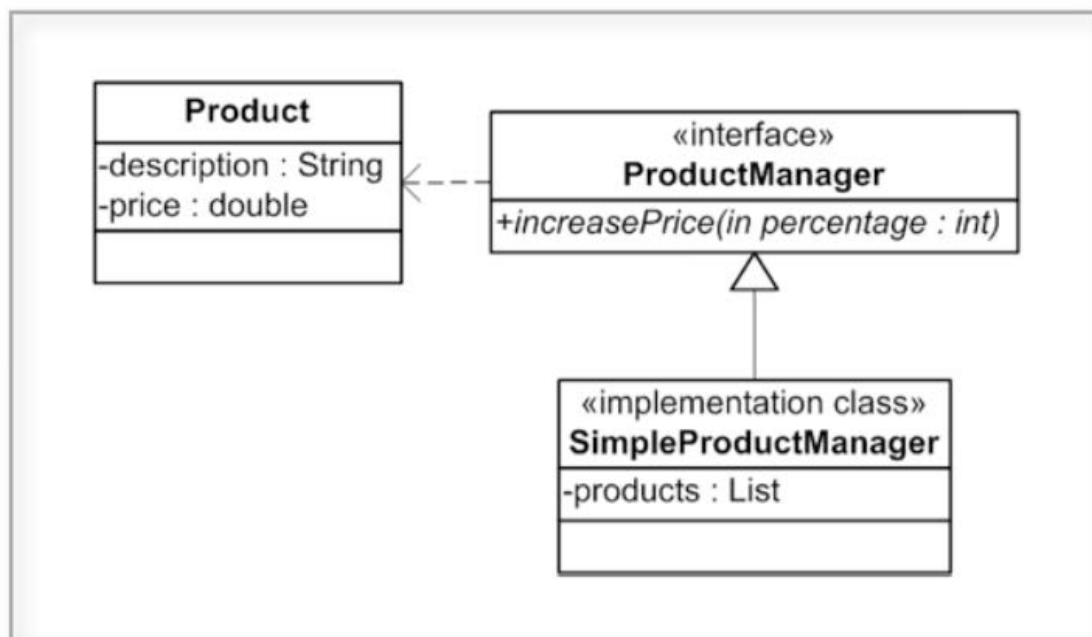


Figure 16. *Diagrama de clases.*

3.2. Añadir algunas clases a la lógica de negocio

Añadamos ahora más lógica de negocio en la forma de una clase `Product` y un servicio al que llamaremos `ProductManager` que gestionará todos los productos. Para separar la lógica de la web de la lógica 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'.

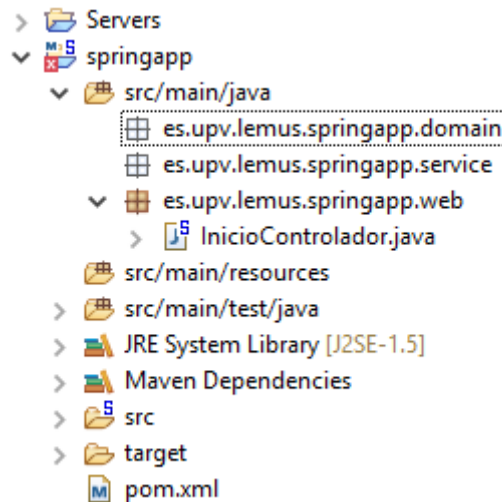


Figure 17. Creación de paquetes para las clases y los servicios.

Primero implementamos la clase `Product` como un POJO con un constructor por defecto (que es provisto si no especificamos ningun constructor explícitamente), así como métodos getters y setters para las propiedades 'description' y 'price'. Además, haremos que la clase implemente la interfaz `Serializable`, aspecto no necesario en este momento para nuestra aplicación, pero que será necesario más tarde cuando persistamos y almacenemos su estado. Esta clase es un objeto de dominio, por lo tanto pertenece al paquete 'domain'.

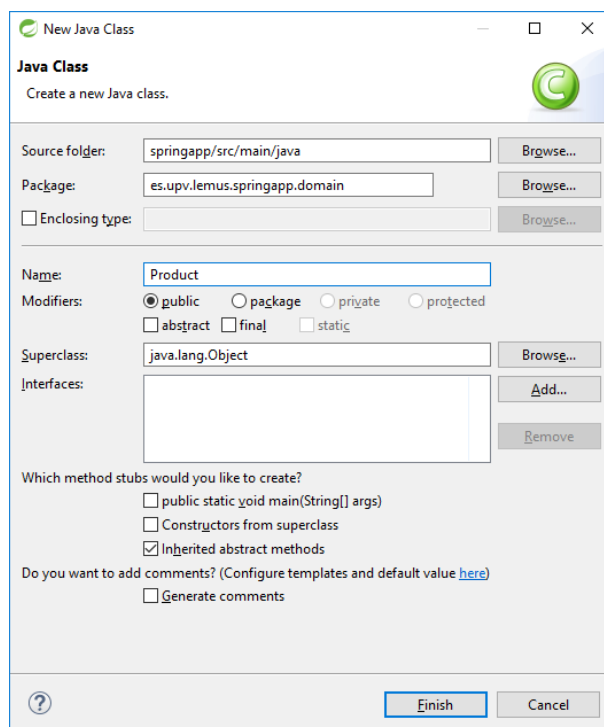


Figure 18. Creación de la clase Product.


```

1 package es.upv.lemus.springapp.domain;
2
3 import java.io.Serializable;
4
5 public class Product implements Serializable {
6
7     private static final long serialVersionUID = 1L;
8
9     private String description;
10    private Double price;
11
12    public String getDescription() {
13        return description;
14    }
15
16    public void setDescription(String description) {
17        this.description = description;
18    }
19
20    public Double getPrice() {
21        return price;
22    }
23
24    public void setPrice(Double price) {
25        this.price = price;
26    }
27
28    public String toString() {
29        StringBuffer buffer = new StringBuffer();
30        buffer.append("Description: " + description + ";");
31        buffer.append("Price: " + price);
32        return buffer.toString();
33    }
34 }

```

Figure 19. Clase *Producto*.

Escribamos ahora una unidad de test para nuestra clase `Product`. Algunos programadores no se molestan en escribir tests para los getters y setters, también llamado código 'auto-generado'. Normalmente, supone demasiado tiempo enfrascarse en el debate (como este párrafo demuestra) sobre si los getters y setters necesitan ser testeados, ya que son métodos demasiado 'triviales'. Nosotros escribiremos los tests debido a que: a) son triviales de escribir; b) tendremos siempre los tests y preferimos pagar el precio de perder un poco de tiempo por la sola ocasión entre cien en que nos salvemos 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 método getter y setter como una pareja en un test simple. Normalmente, escribirás uno o más métodos de test por cada método de la clase, donde cada uno de estos

métodos compruebe una condición particular en el método de la clase (como por ejemplo, verificar un valor `null` pasado al método, etc.).

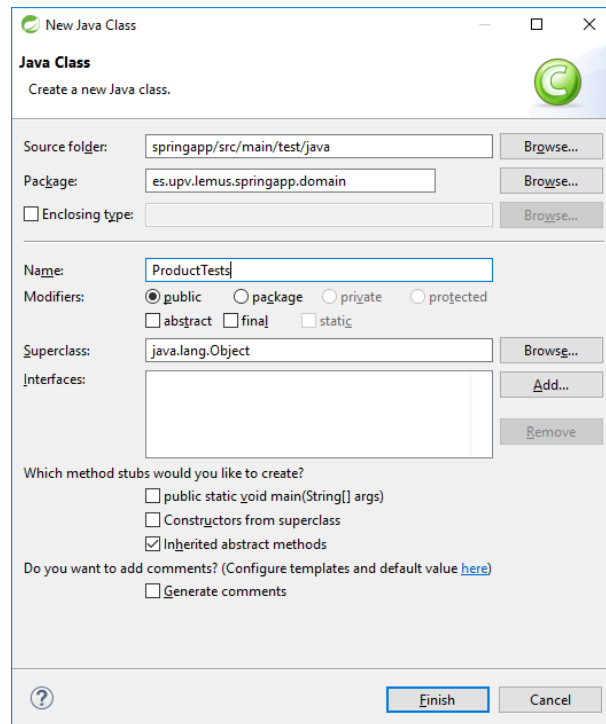


Figure 20. Test Unitario para la clase Producto.

```
1 package es.upv.lemus.springapp.domain;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Before;
6 import org.junit.Test;
7
8 public class ProductTests {
9
10     private Product product;
11
12     @Before
13     public void setUp() throws Exception {
14         product = new Product();
15     }
16
17     @Test
18     public void testSetAndGetDescription() {
19         String testDescription = "aDescription";
20         assertNull(product.getDescription());
21         product.setDescription(testDescription);
22         assertEquals(testDescription, product.getDescription());
23     }
24
25     @Test
26     public void testSetAndGetPrice() {
27         double testPrice = 100.00;
28         assertEquals(0, 0, 0);
29         product.setPrice(testPrice);
30         assertEquals(testPrice, product.getPrice(), 0);
31     }
32 }
33 }
```

Figure 21. Código del Test Unitario para la clase Producto.

A continuación, creamos el servicio `ProductManager`. Éste es el servicio responsable de gestionar los productos. Contiene dos métodos: un método de negocio, `increasePrice()`, que incrementa el precio de todos los productos, y un método 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 más fácil escribir tests unitarios para los `Controllers` (como veremos en el próximo capítulo). Segundo, el uso de interfaces implica que JDK Proxying (una característica del lenguaje Java) puede ser usada para hacer el servicio transaccional, en lugar de usar CGLIB (una librería de generación de código).

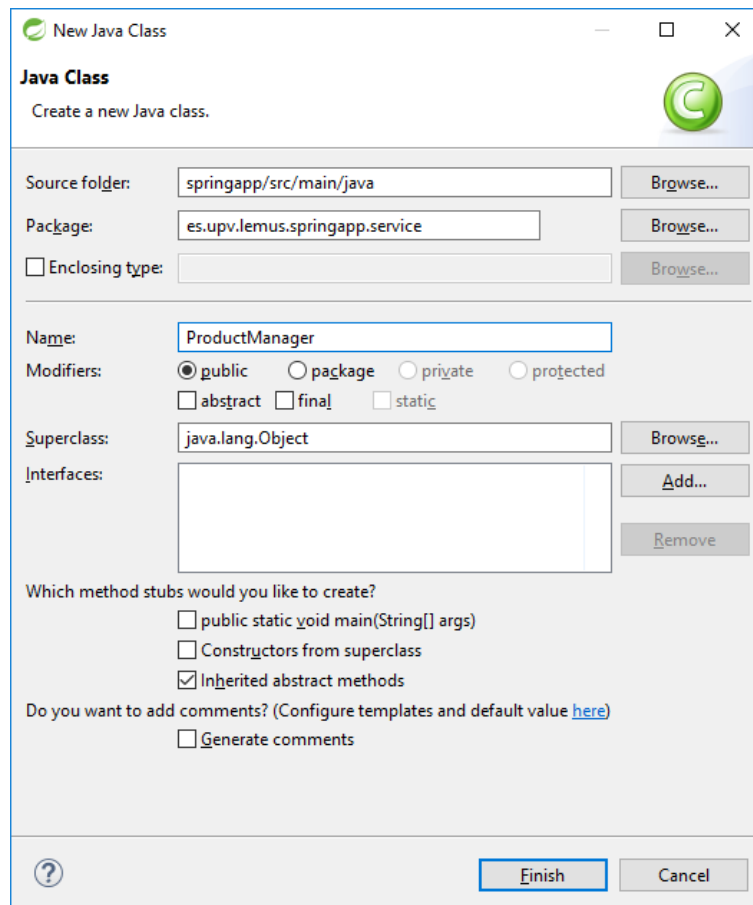


Figure 22. Creación de la interfaz `ProductManager`.

```
1 package es.upv.lemus.springapp.service;
2 import java.io.Serializable;
3 import java.util.List;
4 import es.upv.lemus.springapp.domain.Product;
5
6 public interface ProductManager extends Serializable {
7     public void increasePrice(int percentage);
8     public List<Product> getProducts();
9 }
10
11
```

Figure 23 Código de la interfaz ProductManager.

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

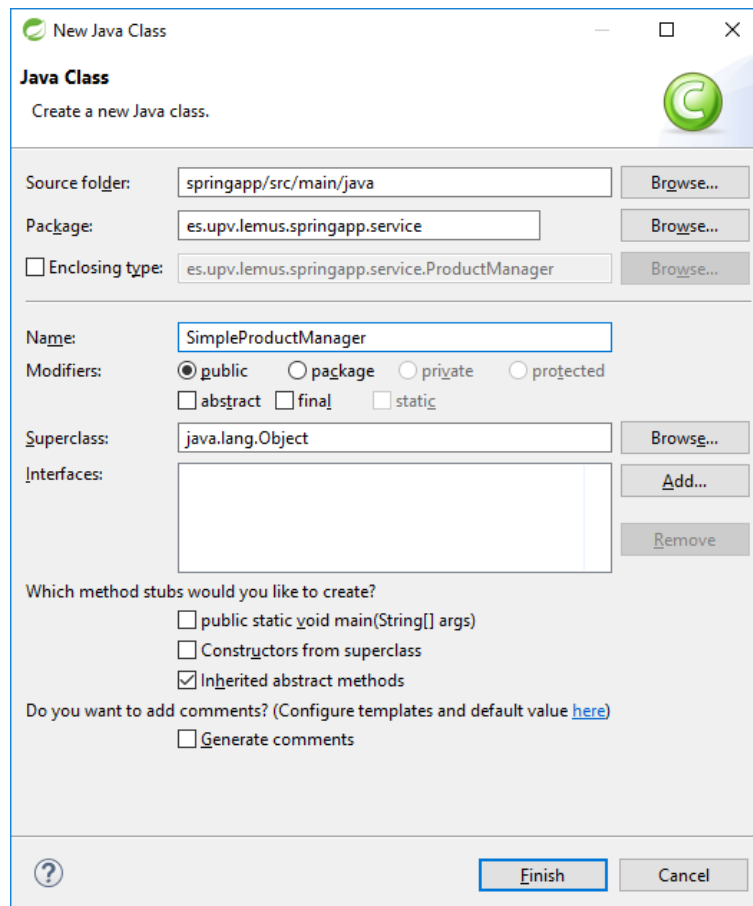
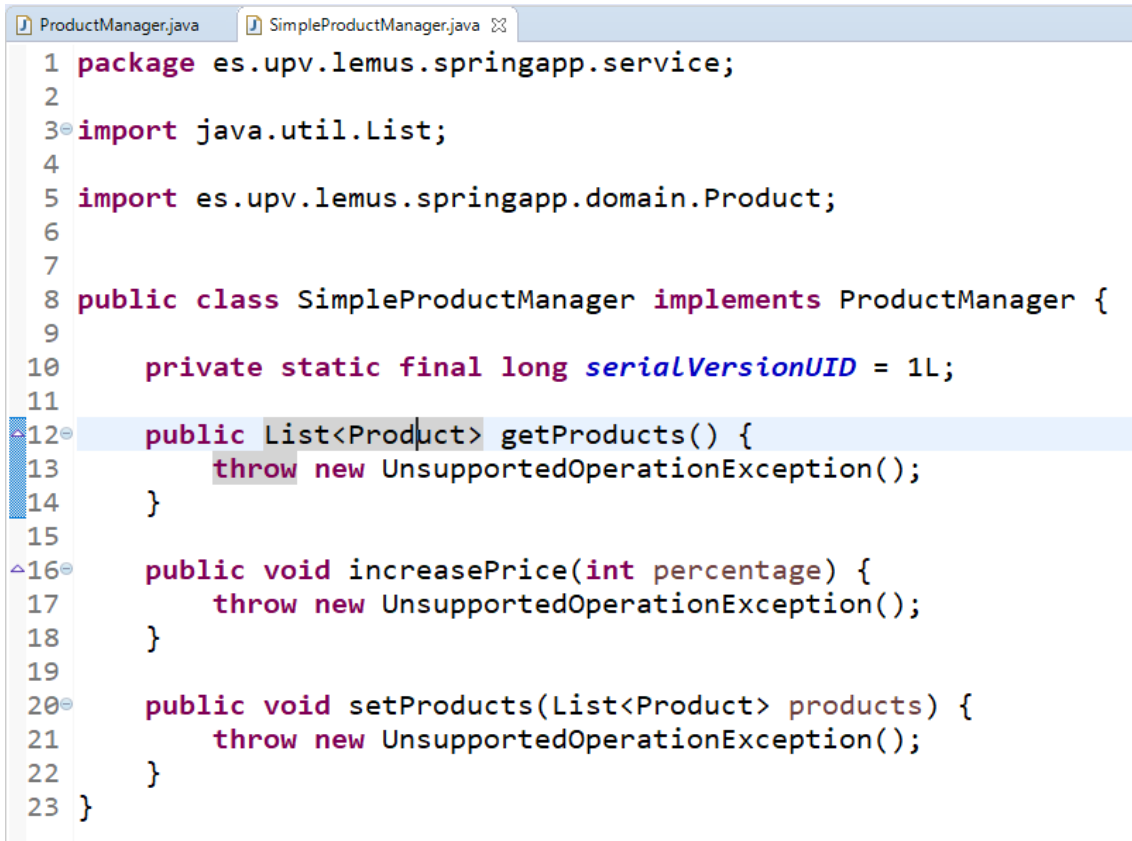


Figure 24 Código de la interfaz ProductManager.

Implementación



```


1 package es.upv.lemus.springapp.service;
2
3 import java.util.List;
4
5 import es.upv.lemus.springapp.domain.Product;
6
7
8 public class SimpleProductManager implements ProductManager {
9
10     private static final long serialVersionUID = 1L;
11
12     public List<Product> getProducts() {
13         throw new UnsupportedOperationException();
14     }
15
16     public void increasePrice(int percentage) {
17         throw new UnsupportedOperationException();
18     }
19
20     public void setProducts(List<Product> products) {
21         throw new UnsupportedOperationException();
22     }
23 }

```

Figure 25 Código de la clase SimpleProductManager que implementa la interfaz ProductManager.

Antes de implementar los métodos en SimpleProductManager, vamos a definir algunos tests. La definición más estricta de Test Driven Development (TDD) implica escribir siempre los tests primero, y a continuación el código. Una interpretación aproximada se conoce como Test Oriented Development (TOD - Desarrollo Orientado a Tests), donde se alternan las tareas de escribir el código y los tests como parte del proceso de desarrollo. En cualquier caso, lo más importante es tener para el código base el conjunto más completo de tests que sea posible. La forma de alcanzar este objetivo es más teoría que práctica. Muchos programadores TDD, sin embargo, están de acuerdo en que la calidad de los tests es siempre mayor cuando son escritos al mismo tiempo que el código, por lo que ésta es la aproximación que vamos a tomar.

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

 New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Figure 26 Creación de la clase SimpleProductManagerTests.

```

1 package es.upv.lemus.springapp.service;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Before;
6 import org.junit.Test;
7
8 public class SimpleProductManagerTests {
9
10     private SimpleProductManager productManager;
11
12     @Before
13     public void setUp() throws Exception {
14         productManager = new SimpleProductManager();
15     }
16
17     @Test
18     public void testGetProductsWithNoProducts() {
19         productManager = new SimpleProductManager();
20         assertNull(productManager.getProducts());
21     }
22 }
23

```

Figure 27 Versión inicial de la Clase SimpleProductManagerTests.

Si ejecutamos ahora los tests de SimpleProductManagerTests fallarán; ya que, por ejemplo, getProducts() todavía no ha sido implementado. Normalmente, es una buena idea marcar los métodos aún no implementados haciendo que lancen una excepción de tipo UnsupportedOperationException.

A continuación, vamos a implementar un test para recuperar una lista de objetos de respaldo en los que han sido almacenados datos de prueba. Sabemos que tenemos que almacenar la lista de productos en la mayoría de nuestros tests de SimpleProductManagerTests, por lo que definimos la lista de objetos de respaldo en el método setUp() de JUnit. Este método, anotado como @Before, será invocado previamente a cada llamada a un método de test.

```

package es.upv.lemus.springapp.service;
import java.util.ArrayList;
import java.util.List;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import es.upv.lemus.springapp.domain.Product;
public class SimpleProductManagerTests {
    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);

@Before
public 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);
}

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

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

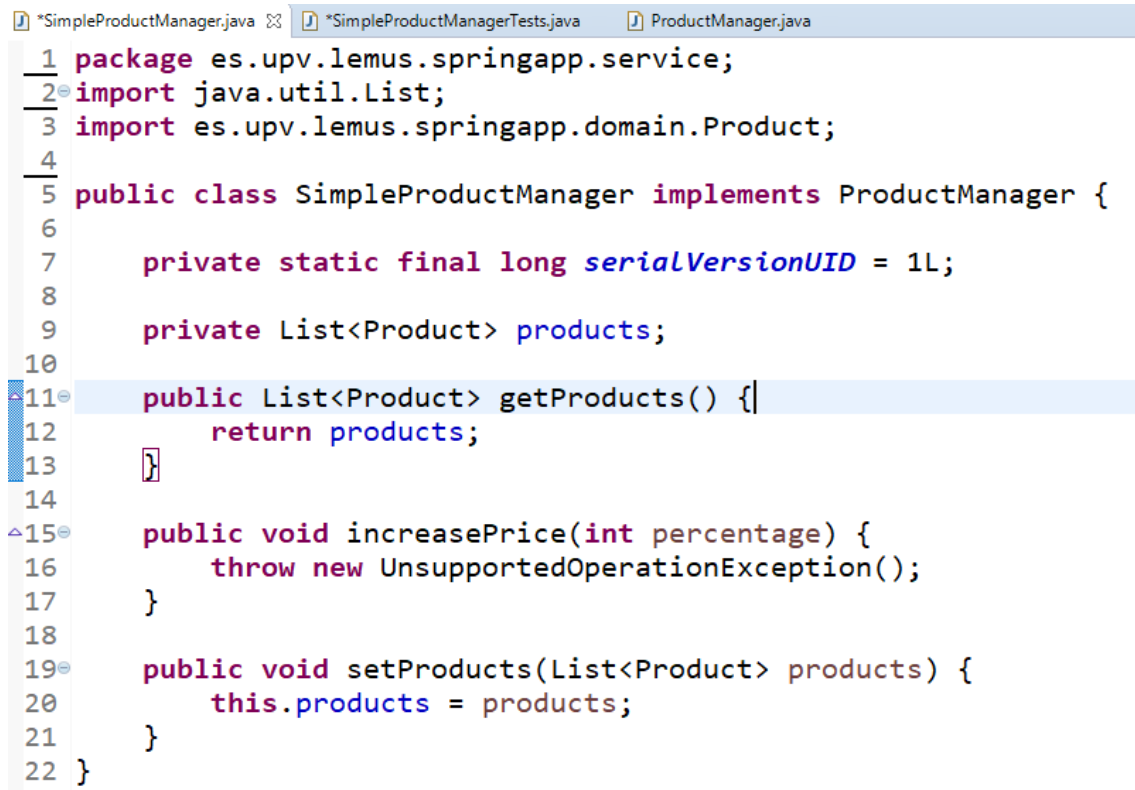
    Product product = products.get(0);
    assertEquals(CHAIR_DESCRIPTION, product.getDescription());
    assertEquals(CHAIR_PRICE, product.getPrice());

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

```

Figure 28 Modificación de la Clase SimpleProductManagerTests.

Si volvemos a lanzar los test de SimpleProductManagerTests seguirán fallando. Para solucionarlo, volvemos a SimpleProductManager e implementamos los métodos getter y setter de la propiedad products.



```

1 package es.upv.lemus.springapp.service;
2 import java.util.List;
3 import es.upv.lemus.springapp.domain.Product;
4
5 public class SimpleProductManager implements ProductManager {
6
7     private static final long serialVersionUID = 1L;
8
9     private List<Product> products;
10
11     public List<Product> getProducts() {
12         return products;
13     }
14
15     public void increasePrice(int percentage) {
16         throw new UnsupportedOperationException();
17     }
18
19     public void setProducts(List<Product> products) {
20         this.products = products;
21     }
22 }

```

Figure 29 Modificación de la clase SimpleProductManager.

Relanza los test de nuevo y ahora todos ellos deben pasar.

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

- La lista de productos es null y el método se ejecuta correctamente.
- La lista de productos está vacía y el método 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.

```

package es.upv.lemus.springapp.service;
import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import es.upv.lemus.springapp.domain.Product;

public class SimpleProductManagerTests {

    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;

@Before
public 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);
}

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

@Test
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());
}

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

@Test
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.");
    }
}

@Test
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(), 0);

    product = products.get(1);
    assertEquals(expectedTablePriceWithIncrease, product.getPrice(), 0);
}
}

```

Figure 30 Versión final de la clase SimpleProductManagerTests.

Por último, volvemos a `SimpleProductManager` para implementar el método `increasePrice()`.

Si volvemos a lanzar los tests de `SimpleProductManagerTests` ahora deberán pasar todos. JUnit tiene un dicho: "keep the bar green to keep the code clean (mantén la barra verde para mantener el código limpio)".

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

3.3. Resumen

Echemos un rápido 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 unitarios para todas las clases que hemos implementado.
- No hemos escrito ni una sola línea de código de Spring. Éste es un ejemplo de lo no-intrusivo que es realmente Spring Framework. Uno de sus propósitos principales es permitir a los programadores centrarse en la parte más importante de todas: modelar e implementar requerimientos de negocio. Otro de sus propósitos es hacer seguir las mejores prácticas de programación de una manera sencilla, como por ejemplo implementar servicios usando interfaces y usar tests unitarios

más allá de las obligaciones pragmáticas de un proyecto dado. A lo largo de este tutorial, verás cómo los beneficios de diseñar interfaces cobran vida.

A continuación, puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.

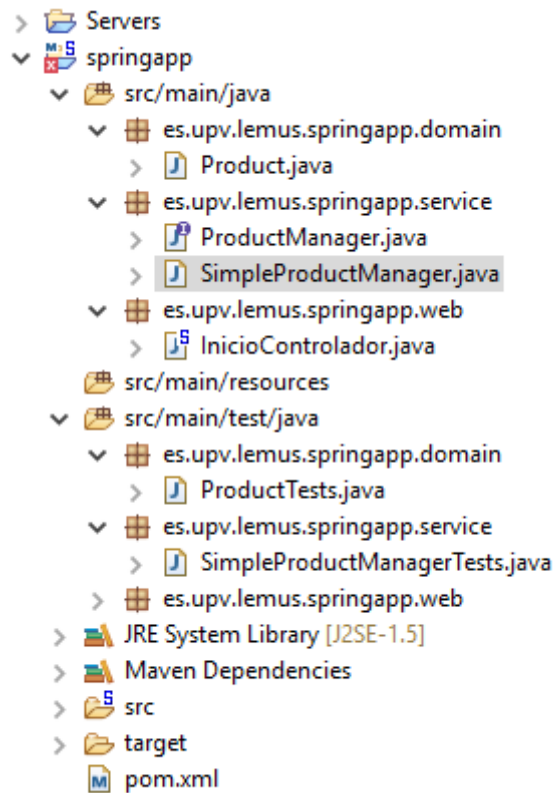


Figure 31 Estructura de ficheros del proyecto.

Capítulo 4. Desarrollando la Interface Web

Ésta es la Parte 4 del tutorial paso a paso para desarrollar una aplicación web desde cero usando Spring Framework. En la [Parte 1](#) hemos configurado el entorno y montado la aplicación básica. En la [Parte 2](#) hemos mejorado la aplicación que habíamos construido hasta entonces. La [Parte 3](#) añade toda la lógica de negocio y los tests unitarios. Ahora es el momento de construir la interface web para la aplicación.

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

Para empezar, renombramos el controlador `InicioController` a algo más descriptivo, como por ejemplo `InventoryController`, puesto que estamos construyendo un sistema de inventario. Aquí es donde un IDE con opción de refactorizar es de valor incalculable. Renombramos `InicioController` a `InventoryController` así como `InicioControladorTests` a `InventoryControllerTests`.

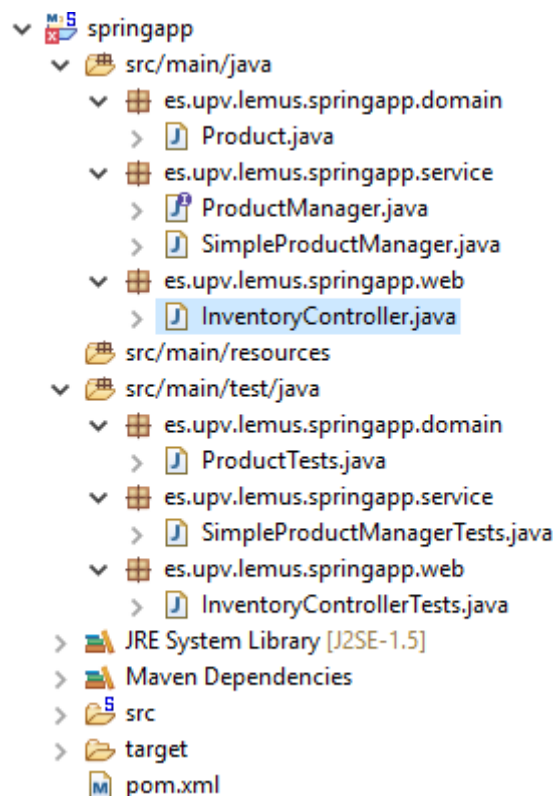


Figure 32 Renombrado de las clases `InicioControlador` por `InventoryController` y `InicioControladorTests` por `InventoryControllerTests`.

A continuación, modificamos `InventoryController` para que almacene una referencia a la clase `ProductManager`. Anotaremos la referencia con `@Autowired` para que Spring la pueda inyectar automáticamente cuando detecte el componente.

También añadimos código para permitir al controlador pasar la información sobre los productos a la vista. El método `getModelAndView()` ahora devuelve tanto un `Map` con la fecha y hora como una lista de productos.

```

1 package es.upv.lemus.springapp.web;
2
3 import java.io.IOException;
4
5 @Controller
6 public class InventoryController {
7     transient private Logger logger = LoggerFactory.getLogger(InventoryController.class);
8     @Autowired
9     private ProductManager productManager;
10
11     @RequestMapping(value="/inicio.htm")
12     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
13         throws ServletException, IOException {
14
15         String now = (new Date()).toString();
16         logger.info("Returning inicio view with " + now);
17
18         Map<String, Object> myModel = new HashMap<String, Object>();
19         myModel.put("now", now);
20         myModel.put("products", this.productManager.getProducts());
21
22         return new ModelAndView("inicio", "model", myModel);
23     }
24
25     public void setProductManager(ProductManager productManager) {
26         this.productManager = productManager;
27     }
28 }

```

Figure 33 Controlador InventoryController.

Antes de que los test puedan ser pasados de nuevo, también necesitaremos modificar `InventoryControllerTest` para que proporcione un `ProductManager` y extraiga el valor de 'now' desde el modelo `Map`.

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 sección que muestra información de cada producto. También vamos a reemplazar el título, la cabecera y el texto de bienvenida con una etiqueta JSTL `<fmt:message/>` que extrae el texto a mostrar desde una ubicación 'message' – veremos esta ubicación un poco más adelante.

```

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

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

```

Figure 34 Código modificado de '`springapp/src/main/webapp/WEB-INF/views/inicio.jsp`'.

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <%@ include file="/WEB-INF/views/include.jsp" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
8   <meta name="author" content="Lenin Lemus">
9   <meta name="viewport" content="width=device-width, initial-scale=1.0">
10  <title>Curso Spring <fmt:message key="title"/></title>
11 </head>
12 <body>
13   <h1>Curso Spring 2018 <c:out value="${now}"/></h1>
14
15   <h1><fmt:message key="heading"/></h1>
16   <p><fmt:message key="greeting"/> <c:out value="${model.now}"/></p>
17   <h3>Products</h3>
18   <c:forEach items="${model.products}" var="prod">
19     <c:out value="${prod.description}"/> <i><c:out value="${prod.price}"/></i><br><br>
20   </c:forEach>
21
22   <p>2018 &copy;Lenin Lemus</p>
23 </body>
24 </html>
```

Figure 35 Modificaciones al código de la página inicio.jsp.

La salida es poco impresionante, ya que nos faltan datos.

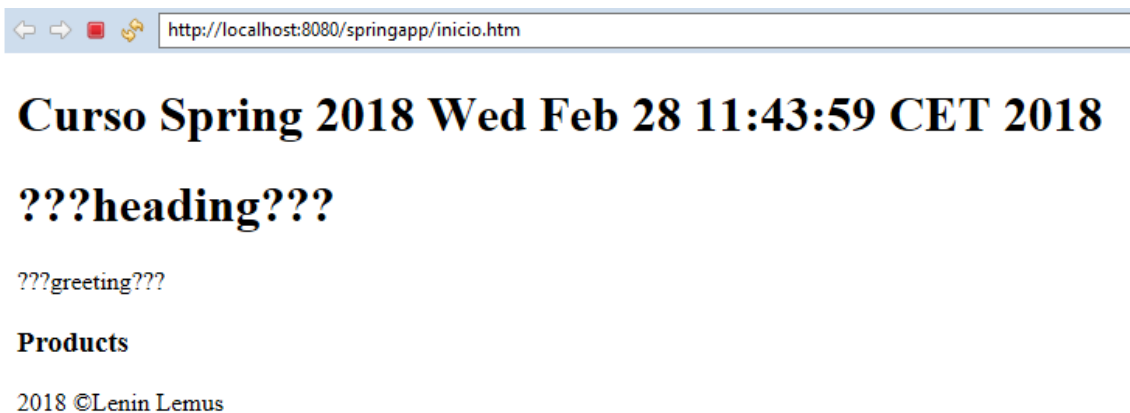


Figure 36 Resultado de la ejecución del programa.

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

Es el momento de añadir un `SimpleProductManager` a nuestro archivo de configuración, el cual se inyectará automáticamente en el `InventoryController`. Todavía no vamos a añadir ningún código 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 en el fichero de configuración de la aplicación. Para ello, simplemente pondremos los datos que necesitamos en un puñado de entradas bean en el archivo '`app-config.xml`'. También añadiremos el bean '`messageSource`'

que nos permitirá recuperar mensajes desde la ubicación **'messages.properties'**, que crearemos en el próximo paso.

```
app-config.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
7       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
8       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9
10     <bean id="productManager" class="es.upv.lemus.springapp.service.impl.SimpleProductManager">
11       <property name="products">
12         <list>
13           <ref bean="product1"/>
14           <ref bean="product2"/>
15           <ref bean="product3"/>
16         </list>
17       </property>
18     </bean>
19
20     <bean id="product1" class="es.upv.lemus.springapp.domain.Product">
21       <property name="description" value="Lamp"/>
22       <property name="price" value="5.75"/>
23     </bean>
24
25     <bean id="product2" class="es.upv.lemus.springapp.domain.Product">
26       <property name="description" value="Table"/>
27       <property name="price" value="75.25"/>
28     </bean>
29
30     <bean id="product3" class="es.upv.lemus.springapp.domain.Product">
31       <property name="description" value="Chair"/>
32       <property name="price" value="22.79"/>
33     </bean>
34
35     <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
36       <property name="basename" value="messages"/>
37     </bean>
38     <!-- Scans the classpath of this application for @Components to deploy as beans -->
39     <context:component-scan base-package="es.upv.lemus.springapp.web" />
40
41     <!-- Configures the @Controller programming model -->
42     <mvc:annotation-driven/>
43     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
44       <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"></property>
45       <property name="prefix" value="/WEB-INF/views/"></property>
46       <property name="suffix" value=".jsp"></property>
47     </bean>
48 </beans>
```

Figure 37 Datos para la aplicación.

4.4. Añadir una ubicación para los mensajes

Creamos un archivo llamado **'messages.properties'** en el directorio **'src/main/webapp/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 **'inicio.jsp'**.

```
title=SpringApp
heading=Inicio :: SpringApp
greeting=Curso de Spring
```

Si ahora compilamos y desplegamos de nuevo la aplicación en el servidor, deberíamos ver lo siguiente en el navegador:

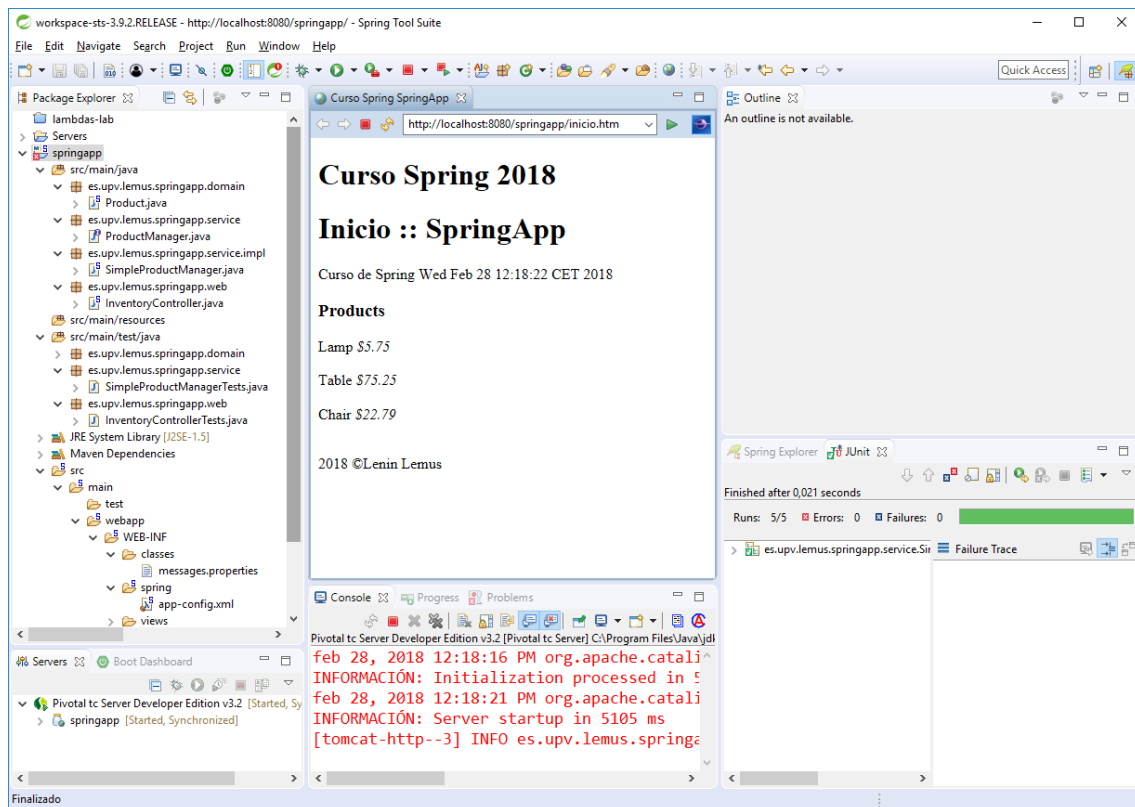


Figure 38 Aspecto de la aplicación, con los datos declarados.

4.5. Añadir un formulario

Para proveer de una interface a la aplicación web que muestre la funcionalidad para incrementar los precios, vamos a añadir un formulario que permitirá al usuario introducir un valor de porcentaje. Para ello, creamos el archivo JSP `'priceincrease.jsp'` en el directorio `'src/main/webapp/WEB-INF/views'`.

```

priceincrease.jsp
1 <%@ include file="/WEB-INF/views/include.jsp" %>
2 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
3
4 <html>
5 <head>
6   <title><fmt:message key="title"/></title>
7   <style>
8     .error { color: red; }
9   </style>
10 </head>
11 <body>
12 <h1><fmt:message key="priceincrease.heading"/></h1>
13 <form:form method="post" commandName="priceIncrease">
14   <table >
15     <tr>
16       <td align="right" width="20%">Increase (%):</td>
17       <td width="20%">
18         <form:input path="percentage"/>
19       </td>
20       <td width="60%">
21         <form:errors path="percentage" cssClass="error"/>
22       </td>
23     </tr>
24   </table>
25   <br>
26   <input type="submit" value="Execute">
27 </form:form>
28 <a href="<c:url value="hello.htm"/>">Home</a>
29 </body>
30 </html>

```

Figure 39 Código del formulario para incrementar precios.

A continuación, debemos incluir las siguientes dependencias en el fichero 'pom.xml':

Group Id	Artifact Id	Versión
javax.validation	validation-api	1.1.0.Final
org.hibernate	hibernate-validator	5.2.4.final
org.slf4j	slf4j-api	1.7.21
org.slf4j	slf4j-log4j12	1.7.21

Para configurar adecuadamente la librería log4j y evitar así algunos warnings, recomendamos añadir (si no existiera) una nueva carpeta de recursos de tipo Source folder en nuestro proyecto, a la que llamaremos 'src/main/resources'. Dentro de esta carpeta crearemos los ficheros 'log4j.dtd' y 'log4j.xml' que se muestran a continuación:

Código del fichero log4j.dtd

```

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

<!-- Authors: Chris Taylor, Ceki Gulcu. -->

```

```

<!-- Version: 1.2 -->

<!-- A configuration element consists of optional renderer
elements, appender elements, categories and an optional root
element. -->

<!ELEMENT log4j:configuration (renderer*, appender*,(category|logger)*,root?,
                             categoryFactory?)>

<!-- The "threshold" attribute takes a level value such that all -->
<!-- logging statements with a level equal or below this value are -->
<!-- disabled. -->

<!-- Setting the "debug" enable the printing of internal log4j logging -->
<!-- statements. -->

<!-- By default, debug attribute is "null", meaning that we not do touch -->
<!-- internal log4j logging settings. The "null" value for the threshold -->
<!-- attribute can be misleading. The threshold field of a repository -->
<!-- cannot be set to null. The "null" value for the threshold attribute -->
<!-- simply means don't touch the threshold field, the threshold field -->
<!-- keeps its old value. -->

<!ATTLIST log4j:configuration
  xmlns:log4j          CDATA #FIXED "http://jakarta.apache.org/log4j/"
  threshold             (all|debug|info|warn|error|fatal|off|null) "null"
  debug                (true|false|null) "null"
>

<!-- renderer elements allow the user to customize the conversion of -->
<!-- message objects to String. -->

<!ELEMENT renderer EMPTY>
<!ATTLIST renderer
  renderedClass CDATA #REQUIRED
  renderingClass CDATA #REQUIRED
>

<!-- Appenders must have a name and a class. -->
<!-- Appenders may contain an error handler, a layout, optional parameters -->
<!-- and filters. They may also reference (or include) other appenders. -->
<!ELEMENT appender (errorHandler?, param*, layout?, filter*, appender-ref*)>
<!ATTLIST appender
  name          ID          #REQUIRED
  class CDATA          #REQUIRED
>

<!ELEMENT layout (param*)>
<!ATTLIST layout
  class CDATA          #REQUIRED
>

<!ELEMENT filter (param*)>
<!ATTLIST filter
  class CDATA          #REQUIRED
>

<!-- ErrorHandlers can be of any class. They can admit any number of -->
<!-- parameters. -->

<!ELEMENT errorHandler (param*, root-ref?, logger-ref*, appender-ref*)>
<!ATTLIST errorHandler
  class CDATA          #REQUIRED
>

<!ELEMENT root-ref EMPTY>

<!ELEMENT logger-ref EMPTY>
<!ATTLIST logger-ref
  ref IDREF #REQUIRED
>

<!ELEMENT param EMPTY>
<!ATTLIST param
  name CDATA          #REQUIRED
  value CDATA          #REQUIRED

```

```

>

<!-- The priority class is org.apache.log4j.Level by default -->
<!ELEMENT priority (param*)>
<!ATTLIST priority
  class    CDATA    #IMPLIED
  value    CDATA    #REQUIRED
>

<!-- The level class is org.apache.log4j.Level by default -->
<!ELEMENT level (param*)>
<!ATTLIST level
  class    CDATA    #IMPLIED
  value    CDATA    #REQUIRED
>

<!-- If no level element is specified, then the configurator MUST not -->
<!-- touch the level of the named category. -->
<!ELEMENT category (param*,(priority|level)?,appender-ref*)>
<!ATTLIST category
  class          CDATA    #IMPLIED
  name           CDATA    #REQUIRED
  additivity     (true|false) "true"
>

<!-- If no level element is specified, then the configurator MUST not -->
<!-- touch the level of the named logger. -->
<!ELEMENT logger (level?,appender-ref*)>
<!ATTLIST logger
  name          ID        #REQUIRED
  additivity     (true|false) "true"
>

<!ELEMENT categoryFactory (param*)>
<!ATTLIST categoryFactory
  class          CDATA    #REQUIRED>

<!ELEMENT appender-ref EMPTY>
<!ATTLIST appender-ref
  ref IDREF    #REQUIRED
>

<!-- If no priority element is specified, then the configurator MUST not -->
<!-- touch the priority of root. -->
<!-- The root category always exists and cannot be subclassed. -->
<!ELEMENT root (param*, (priority|level)?, appender-ref*)>

<!-- ===== -->
<!-- A logging event -->
<!-- ===== -->
<!ELEMENT log4j:eventSet (log4j:event*)>
<!ATTLIST log4j:eventSet
  xmlns:log4j    CDATA    #FIXED "http://jakarta.apache.org/log4j/"
  version        (1.1|1.2) "1.2"
  includesLocationInfo (true|false) "true"
>

<!ELEMENT log4j:event (log4j:message, log4j:NDC?, log4j:throwable?,
  log4j:locationInfo?) >

<!-- The timestamp format is application dependent. -->
<!ATTLIST log4j:event
  logger    CDATA    #REQUIRED
  level     CDATA    #REQUIRED
  thread    CDATA    #REQUIRED
  timestamp CDATA    #REQUIRED
>

<!ELEMENT log4j:message (#PCDATA)>
<!ELEMENT log4j:NDC (#PCDATA)>

<!ELEMENT log4j:throwable (#PCDATA)>

<!ELEMENT log4j:locationInfo EMPTY>
<!ATTLIST log4j:locationInfo

```

```

class CDATA      #REQUIRED
method CDATA     #REQUIRED
file CDATA       #REQUIRED
line CDATA       #REQUIRED
>

```

Contenido del fichero log4j.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <!-- Appenders -->
    <appender name="console" class="org.apache.log4j.ConsoleAppender">
        <param name="Target" value="System.out" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p: %c - %m%n" />
        </layout>
    </appender>

    <!-- Application logger -->
    <logger name="springapp">
        <level value="info" />
    </logger>

    <!-- 3rdparty Loggers -->
    <logger name="org.springframework.beans">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.jdbc">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.transaction">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.orm">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.web">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.webflow">
        <level value="warn" />
    </logger>

    <!-- Root Logger -->
    <root>
        <priority value="warn" />
        <appender-ref ref="console" />
    </root>

</log4j:configuration>

```

La siguiente clase que crearemos es un JavaBean muy sencillo que solamente contiene una propiedad, con sus correspondientes métodos getter y setter. Éste es el objeto que el formulario rellenará y desde el que nuestra lógica de negocio extraerá el porcentaje de incremento que queremos aplicar a los precios. La clase `PriceIncrease` utiliza las anotaciones `@Min` y `@Max` para definir el intervalo de valores válido para el incremento de precios del stock.

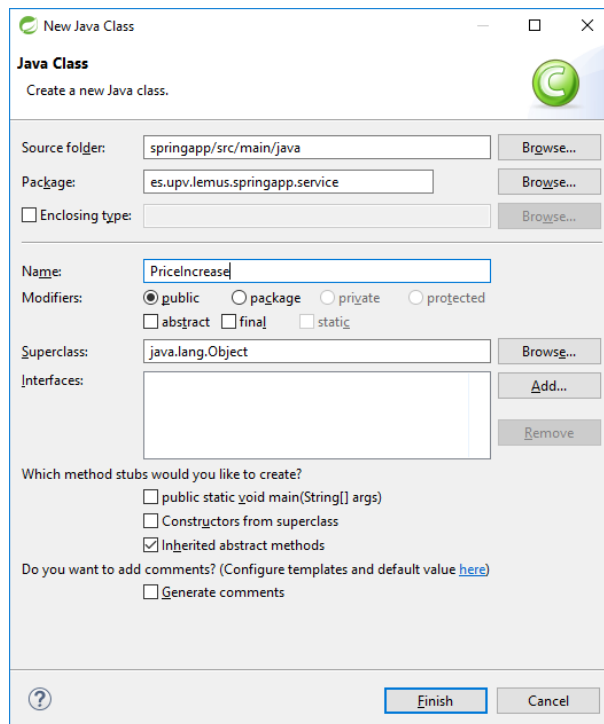


Figure 40 Creación s de la clase PriceIncrease.

4.6. Añadir un controlador de formulario

A continuación, creamos la clase `PriceIncreaseFormController` que actuará como controlador de las peticiones de incremento de precio realizadas desde el formulario. Spring inyectará automáticamente al controlador del formulario la referencia al servicio `ProductManager` gracias a la anotación `@Autowired`. El método `formBackingObject(..)` será invocado antes de que el formulario se muestre al usuario (petición GET) y rellenará el campo con un incremento por defecto de un 15%. El método `onSubmit(..)` será invocado cuando el usuario envíe del formulario a través del método POST. El uso de la anotación `@Valid` permitirá validar el incremento introducido y volverá a mostrar el formulario en caso de que éste no sea válido.

Para mostrar los distintos mensajes de error, vamos a añadir también algunos mensajes al archivo de mensajes `'messages.properties'`.

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

<html>
  <head><title><fmt:message key="title"/></title></head>
  <body>
    <h1><fmt:message key="heading"/></h1>
    <p><fmt:message key="greeting"/> <c:out
value="${model.now}"/></p>
    <h3>Products</h3>

    <c:forEach items="${model.products}" var="prod">
      <c:out value="${prod.description}"/> <i>${<c:out
value="${prod.price}"/></i><br><br>
```

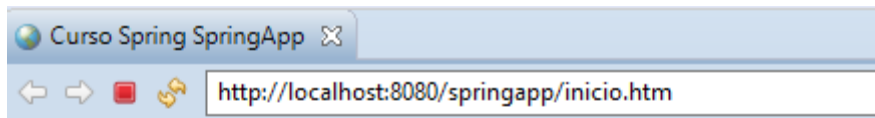
```
</c:forEach>
<br>
<a href="<c:url value="priceincrease.htm"/>">Increase Prices</a>
<br>
</body>
</html>
```

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 recuperar una lista de productos para mostrar.
- Entonces hemos definido algunos datos de prueba para rellenar objetos de negocio.
- A continuación, hemos modificado la página JSP para usar una ubicación de mensajes y hemos añadido un loop `forEach` para mostrar una lista dinámica de productos.
- Después hemos creado un formulario para disponer de la capacidad de incrementar los precios.
- Finalmente hemos creado un controlador de formulario que valida los datos introducidos, hemos desplegado y probado las nuevas características.

A continuación, puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.



Curso Spring 2018

Inicio :: SpringApp

Curso de Spring Wed Feb 28 15:00:19 CET 2018

Products

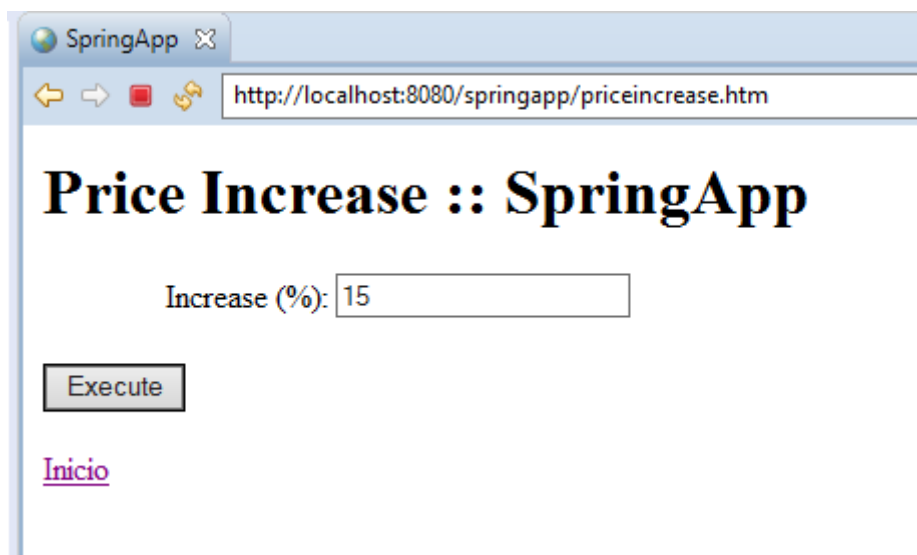
Lamp \$5.75

Table \$75.25

Chair \$22.79

[Increase Prices](#)

2018 ©Lenin Lemus





Capítulo 5. Implementando Persistencia en Base de Datos

Esta es la Parte 5 del tutorial paso a paso sobre cómo desarrollar una aplicación web desde cero usando Spring Framework.

En la [Parte 1](#) hemos configurado el entorno y puesto en marcha una aplicación básica.

En la [Parte 2](#) hemos mejorado la aplicación que habíamos construido hasta entonces.

En la [Parte 3](#) hemos añadido toda la lógica de negocio y los tests unitarios,

y en la [Parte 4](#) hemos desarrollado la interface web.

Ahora es el momento de introducir persistencia en base de datos. En las partes anteriores hemos visto cómo cargar algunos objetos de negocio definiendo beans en un archivo de configuración.

Es obvio que esta solución nunca va a funcionar en el mundo real – cada vez que reiniciemos el servidor obtendremos de nuevo los precios originales. Necesitamos añadir código para persistir esos cambios en una base de datos.

5.1. Creación y relleno de la base de datos

Antes de que podamos comenzar a desarrollar el código de persistencia, necesitamos una base de datos. En lugar de confiar en una base de datos integrada con la propia aplicación, vamos a utilizar una base de datos separada.

Hemos planeado usar MySQL, una buena base de datos de código libre. Sin embargo, los pasos que se muestran a continuación serán similares para otras bases de datos (p. ej. PostgreSQL, HSQL...). Desde el enlace anterior, se puede descargar MySQL para diferentes sistemas operativos. Una vez instalada, cada distribución proporciona sus correspondientes scripts de inicio para arrancar la base de datos.

Primero, creamos el fichero **'springapp.sql'** (en el directorio **'db'**) con las sentencias SQL necesarias para la creación de la base de datos `springapp`. Este fichero creará también la tabla `products`, que alojará los productos de nuestra aplicación. Además, establecerá los permisos para el usuario `springappuser`.

'springapp/db/springapp.sql':

```
CREATE DATABASE springapp;

GRANT ALL ON springapp.* TO springappuser@'%' IDENTIFIED BY
'pspringappuser';
GRANT ALL ON springapp.* TO springappuser@localhost IDENTIFIED BY
'pspringappuser';

USE springapp;

CREATE TABLE products (
  id INTEGER PRIMARY KEY,
  description varchar(255),
  price decimal(15,2)
);
CREATE INDEX products_description ON products(description);
```

A continuación, necesitamos añadir nuestros datos de prueba.

Para ello, creamos 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);
```

Por último, ejecutamos las siguientes instrucciones sobre [la línea de comandos](#) para crear y rellenar la base de datos:

```
springapp/db# mysql -u root -p
Enter password:
...
mysql> source springapp.sql
mysql> source load_data.sql
```

```
C:\Users\tefra\Documents\CursoSpring\springapp-05\springapp\db>mysql -u dbuser -p
Enter password: *****
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 108
Server version: 10.1.28-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> source springapp.sql
Query OK, 1 row affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Database changed
Query OK, 0 rows affected (0.03 sec)

Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

MariaDB [springapp]> source load_data.sql
Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.01 sec)

MariaDB [springapp]>
```

En el fichero '**pom.xml**', para poder acceder desde nuestra aplicación a la base de datos MySQL mediante JPA, debemos incluir las siguientes dependencias:

Group Id	Artifact Id	Version
mysql	mysql-connector-java	5.1.38
org.hibernate.javax.persistence	hibernate-jpa-2.0-api	1.0.2.Final
org.hibernate	hibernate-entitymanager	5.1.0.Final
org.springframework	spring-orm	\${springframework.version}
javax.el	Javax.el-api	2.2.4

5.2. Crear una implementación para JPA de un Objeto de Acceso a Datos (DAO)

Comencemos creando un nuevo paquete llamado **'es.upv.lemus.springapp.repository'** que contendrá cualquier clase que sea usada para el acceso a la base de datos. En este paquete vamos a crear un nuevo interface llamado `ProductDao`. Éste será el interface que definirá la funcionalidad de la implementación DAO que vamos a crear - esto nos permitirá elegir en el futuro otra implementación que se adapte mejor a nuestras necesidades (p. ej. JDBC, etc.).

'springapp/src/main/java/es/upv/lemus/springapp/repository/ProductDao.java':

```
package es.upv.lemus.springapp.repository;

import java.util.List;

import es.upv.lemus.springapp.domain.Product;

public interface ProductDao {

    public List<Product> getProductList();

    public void saveProduct(Product prod);

}
```

A continuación, creamos una clase llamada `JPAProductDao` que será la implementación JPA de la interface anterior. Spring permite creación automática de beans de acceso a datos mediante la anotación `@Repository`. Asimismo, Spring reconoce las anotaciones del API estándar JPA. Por ejemplo, la anotación `@Persistence` es utilizada en la clase `JPAProductDao` para inyectar automáticamente el `EntityManager`.

'springapp/src/main/java/es/upv/lemus/springapp/repository/JPAProductDao.java':

```
package es.upv.lemus.springapp.repository;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import es.upv.lemus.springapp.domain.Product;

@Repository(value = "productDao")
public class JPAProductDao implements ProductDao {

    private EntityManager em = null;
```

```

    /**
     * Sets the entity manager.
     */
    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    @Transactional(readOnly = true)
    @SuppressWarnings("unchecked")
    public List<Product> getProductList() {
        return em.createQuery("select p from Product p order by
p.id").getResultList();
    }

    @Transactional(readOnly = false)
    public void saveProduct(Product prod) {
        em.merge(prod);
    }
}

```

Vamos a echarle un vistazo a los dos métodos DAO en esta clase. El primer método, `getProductList()`, ejecuta una consulta usando el `EntityManager`. Para ello incluimos en él una sentencia SQL que obtiene los objetos persistentes de la clase `Product`. El segundo método, `saveProduct()`, también usa el `EntityManager`. Esta vez hacemos un `merge` para almacenar el producto en la base de datos. Ambos métodos se ejecutan de manera transaccional gracias a la anotación `@Transactional`, con la diferencia de que el método `getProductList()` permite la ejecución de diversas consultas de lectura en paralelo.

Llegados a este punto, debemos modificar la clase `Product` para que se persista correctamente. Para ello modificamos el fichero '**Product.java**' y añadimos las anotaciones de JPA que realizan el mapeo entre los campos del objeto y aquellos de la base de datos. Asimismo, hemos añadido el campo `id` para mapear la clave primaria de la tabla `products`.

'springapp/src/main/java/es/upv/lemus/springapp/domain/Product.java':

```

package es.upv.lemus.springapp.domain;

import java.io.Serializable;

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

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

```

```

private static final long serialVersionUID = 1L;

@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer id;

private String description;
private Double price;

public Integer getId()
{
    return id;
}

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

public String getDescription() {
    return description;
}

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

public Double getPrice() {
    return price;
}

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

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

```

Los pasos anteriores completan la implementación JPA en nuestra capa de persistencia.

5.3. Implementar tests para la implementación DAO sobre JPA

Es el momento de añadir tests a nuestra aplicación DAO sobre JPA. Para ello, crearemos la clase **'JPAProductDaoTests'** dentro de paquete **'es.upv.lemus.springapp.repository'** de la carpeta **'src/test/java'**.

'springapp/src/test/java/es/upv/lemus/springapp/repository/JPAProductDaoTests.java':

```
package es.upv.lemus.springapp.repository;

import java.util.List;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

import es.upv.lemus.springapp.domain.Product;

public class JPAProductDaoTests {

    private ApplicationContext context;
    private ProductDao productDao;

    @Before
    public void setUp() throws Exception {
        context = new
ClassPathXmlApplicationContext("classpath:test-context.xml");
        productDao = (ProductDao) context.getBean("productDao");
    }

    @Test
    public void testGetProductList() {
        List<Product> products = productDao.getProductList();
        assertEquals(products.size(), 3, 0);
    }

    @Test
    public void testSaveProduct() {
        List<Product> products = productDao.getProductList();

        Product p = products.get(0);
        Double price = p.getPrice();
        p.setPrice(200.12);
        productDao.saveProduct(p);

        List<Product> updatedProducts = productDao.getProductList();
        Product p2 = updatedProducts.get(0);
        assertEquals(p2.getPrice(), 200.12, 0);

        p2.setPrice(price);
    }
}
```



```

        productDao.saveProduct(p2);
    }
}

```

Aún no disponemos del archivo que contiene el contexto de la aplicación, y que es cargado por este test, por lo que vamos a crear este archivo en una nueva carpeta de recursos de tipo Source folder en nuestro proyecto, a la que llamaremos **'src/test/resources'**:

'springapp/src/test/resources/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"
       xmlns:context="http://www.springframework.org/schema/context"
       "
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- holding properties for database connectivity /-->
    <context:property-placeholder
location="classpath:jdbc.properties"/>

    <!-- enabling annotation driven configuration /-->
    <context:annotation-config/>

    <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="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
        p:dataSource-ref="dataSource"
        p:jpaVendorAdapter-ref="jpaAdapter">
        <property name="loadTimeWeaver">
            <bean
class="org.springframework.instrument.classloading.InstrumentationL
oadTimeWeaver"/>
        </property>
        <property                                name="persistenceUnitName"
value="springappPU"></property>
    </bean>

    <bean id="jpaAdapter"

```

```

        class="org.springframework.orm.jpa.vendor.HibernateJpaVendor
Adapter"
        p:database="${jpa.database}"
        p:showSql="${jpa.showSql}"/>

        <bean                                id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager"
        p:entityManagerFactory-ref="entityManagerFactory"/>

        <tx:annotation-driven                transaction-
manager="transactionManager"/>

        <!-- Scans the classpath of this application for @Components to
deploy as beans -->
        <context:component-scan                base-
package="com.companyname.springapp.repository" />
        <context:component-scan                base-
package="com.companyname.springapp.service" />

</beans>

```

Hemos definido un `productDao` el cual es la clase que estamos testeando. Además hemos definido un `DataSource` con comodines para los valores de configuración. Sus valores serán tomados de un archivo de propiedades en tiempo de ejecución. El bean `property-placeholder` que hemos declarado leerá este archivo de propiedades y sustituirá cada comodín con su valor actual. Esto es conveniente puesto que separa los valores de conexión en su propio archivo, y estos valores a menudo suelen ser cambiados durante el despliegue de la aplicación. Vamos a poner este nuevo archivo tanto en el directorio **'src/test/resources'** como en el directorio **'webapp/WEB-INF/classes'** por lo que estará disponible cuando ejecutemos los tests y cuando desplaguemos la aplicación web. El contenido de este archivo de propiedades es:

'springapp/src/test/resources/jdbc.properties' y
'springapp/src/main/webapp/WEB-INF/classes/jdbc.properties':

```

jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/springapp
jdbc.username=springappuser
jdbc.password=pspringappuser
hibernate.dialect=org.hibernate.dialect.MySQLDialect

jpa.database = MYSQL
hibernate.generate_statistics = true
hibernate.show_sql = true
jpa.showSql = true
jpa.generateDdl = true

```

Por otro lado, la configuración del bean `entityManager` requerirá la creación de un fichero donde se defina la unidad de persistencia. Por defecto, este fichero se deberá llamar **'persistence.xml'** y deberá crearse bajo el directorio **'META-INF'** dentro del directorio de recursos de la aplicación **'src/main/resources'**

'springapp/src/main/resources/META-INF/persistence.xml':

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="springappPU" transaction-
type="RESOURCE_LOCAL">
    </persistence-unit>
</persistence>
```

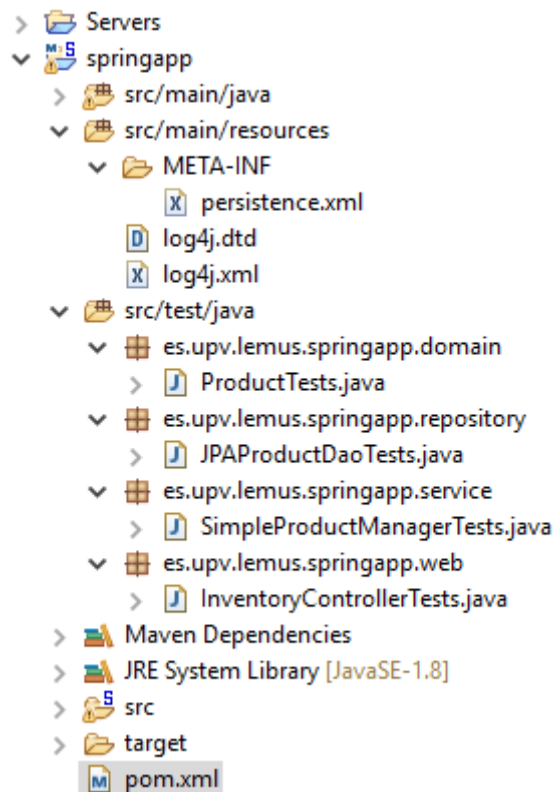
Ahora disponemos del código suficiente para ejecutar los tests de **'JPAProductDaoTests'** y hacerlos pasar.

5.4. Resumen

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

- Primero hemos configurado nuestra base de datos y ejecutado las sentencias SQL para crear una tabla en la base de datos y cargar algunos datos de prueba.
- Hemos creado una clase DAO que manejará el trabajo de persistencia mediante JPA usando la clase `Product`.
- Finalmente hemos creado tests de integración para comprobar su funcionamiento.

A continuación, puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.



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

Capítulo 6. Integrando la Aplicación Web con la Capa de Persistencia

Esta es la Parte 6 del tutorial paso a paso sobre cómo desarrollar una aplicación web desde cero usando Spring Framework.

En la [Parte 1](#) hemos configurado el entorno y puesto en marcha una aplicación básica.

En la [Parte 2](#) hemos mejorado la aplicación que habíamos construido hasta entonces.

En la [Parte 3](#) hemos añadido toda la lógica de negocio y los tests unitarios, y en la [Parte 4](#) hemos desarrollado la interface web.

En la [Parte 5](#) hemos desarrollado la capa de persistencia.

Ahora es el momento de integrarlo todo junto en una aplicación web completa.

6.1. Modificar la Capa de Servicio

Si hemos estructurado nuestra aplicación adecuadamente, sólo 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 deberían ser conscientes de ningún detalle de la implementación de la capa de servicio. Así que vamos a añadir persistencia a la implementación de ProductManager.

Modifica la clase SimpleProductManager y añade una referencia a la interface ProductDao además de un método setter para esta referencia. Qué implementación usemos debe ser irrelevante para la clase ProductManager, a la cual se le inyectará el DAO de manera automática a través del método llamado setProductDao. El método getProducts usará ahora este DAO para recuperar la lista de productos. Finalmente, el método increasePrices recuperará la lista de productos y, después de haber incrementado los precios, almacenará los productos de nuevo en la base de datos usando el método saveProduct definido en el DAO.

'springapp/src/main/java/es/upv/lemus/springapp/service/SimpleProductManager.java':

```
package es.upv.lemus.springapp.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import es.upv.lemus.springapp.domain.Product;
import es.upv.lemus.springapp.repository.ProductDao;

@Component
public class SimpleProductManager implements ProductManager {

    private static final long serialVersionUID = 1L;

    @Autowired
    private ProductDao productDao;

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

    public List<Product> getProducts() {
        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);  
    }  
}  
}
```

6.2. Resolver los tests fallidos

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

'springapp/src/test/java/es/upv/lemus/springapp/repository/InMemoryProductDao.java':

```
package es.upv.lemus.springapp.repository;

import java.util.List;

import es.upv.lemus.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 aquí está la versión modificada de `SimpleProductManagerTests`:

'springapp/src/test/java/es/upv/lemus/springapp/service/SimpleProductManagerTests.java':

```
package es.upv.lemus.springapp.service;

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

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import es.upv.lemus.springapp.domain.Product;
import es.upv.lemus.springapp.repository.InMemoryProductDao;
import es.upv.lemus.springapp.repository.ProductDao;
```



```

public class SimpleProductManagerTests {

    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;

    @Before
    public 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);
    }

    @Test
    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
productManager.setProductDao(new InMemoryProductDao(null));
        assertNull(productManager.getProducts());
    }

    @Test
    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());
    }
}

```

```

    }

    @Test
    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.");
        }
    }

    @Test
    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.");
        }
    }

    @Test
    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(), 0);

        product = products.get(1);
        assertEquals(expectedTablePriceWithIncrease,
product.getPrice(), 0);
    }
}

```

También necesitamos modificar `InventoryControllerTests` puesto que esta clase también usa `SimpleProductManager`. Aquí está la versión modificada de `InventoryControllerTests`:

'springapp/src/test/java/es/upv/lemus/springapp/web/InventoryControllerTests.java':

```
package es.upv.lemus.springapp.web;

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

import static org.junit.Assert.*;

import org.junit.Test;
import org.springframework.web.servlet.ModelAndView;

import es.upv.lemus.springapp.domain.Product;
import es.upv.lemus.springapp.repository.InMemoryProductDao;
import es.upv.lemus.springapp.service.SimpleProductManager;

public class InventoryControllerTests {

    @Test
    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("inicio", modelAndView.getViewName());
        assertNotNull(modelAndView.getModel());
        @SuppressWarnings("unchecked")
        Map modelMap = (Map) modelAndView.getModel().get("model");
        String nowValue = (String) modelMap.get("now");
        assertNotNull(nowValue);
    }
}
```

Una vez realizados los cambios anteriores, comprobad que los tests `SimpleProductManagerTests` y `InventoryControllerTests` se ejecutan satisfactoriamente.

6.3. Crear un nuevo contexto de aplicación para configurar la capa de servicio

Hemos visto antes que es tremendamente fácil modificar la capa de servicio para usar persistencia en base de datos. Esto es así porque está despegada de la capa web. Ahora es el momento de despegar también la configuración de la capa de servicio de la capa web. Eliminaremos la configuración de `productManager` y la lista de productos del archivo de configuración '**app-config.xml**'. Así es como este archivo quedaría ahora:

'springapp/src/main/webapp/WEB-INF/spring/app-config.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:mvc="http://www.springframework.org/schema/mvc"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

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

  <!-- Scans the classpath of this application for @Components to
  deploy as beans -->
  <context:component-scan base-
package="es.upv.lemus.springapp.web" />

  <!-- Configures the @Controller programming model -->
  <mvc:annotation-driven/>

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

</beans>
```

Todavía necesitamos configurar la capa de servicio y lo haremos en nuestro propio archivo de contexto de aplicación. Este archivo se llama '**applicationContext.xml**' y será cargado mediante un servlet listener que definiremos en '**web.xml**'. Todos los bean configurados en este nuevo contexto de aplicación estarán disponibles desde cualquier contexto del servlet.

'springapp/src/main/webapp/WEB-INF/web.xml':

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

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/applicationContext.xml</param-
value>
    </context-param>

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

    <display-name>Springapp</display-name>

    <servlet>
        <servlet-name>springapp</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                /WEB-INF/spring/app-config.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

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

</web-app>
```

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

'springapp/src/main/webapp/WEB-INF/spring/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:context="http://www.springframework.org/schema/context"
    "
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:tx="http://www.springframework.org/schema/tx"
```

```

        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.2.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.2.xsd">

    <!-- holding properties for database connectivity /-->
    <context:property-placeholder
location="classpath:jdbc.properties"/>

    <!-- enabling annotation driven configuration /-->
    <context:annotation-config/>

    <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="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManag
erFactoryBean"
        p:dataSource-ref="dataSource"
        p:jpaVendorAdapter-ref="jpaAdapter">
        <property name="loadTimeWeaver">
            <bean
class="org.springframework.instrument.classloading.InstrumentationL
oadTimeWeaver"/>
        </property>
        <property                                name="persistenceUnitName"
value="springappPU"></property>
    </bean>

    <bean id="jpaAdapter"
class="org.springframework.orm.jpa.vendor.HibernateJpaVendor
Adapter"
        p:database="${jpa.database}"
        p:showSql="${jpa.showSql}"/>

    <bean                                id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager"
        p:entityManagerFactory-ref="entityManagerFactory"/>

    <tx:annotation-driven                                transaction-
manager="transactionManager"/>

    <!-- Scans the classpath of this application for @Components to
deploy as beans -->
    <context:component-scan                                base-
package="es.upv.lemus.springapp.repository" />

```

```
<context:component-scan                                base-  
package="es.upv.lemus.springapp.service" />  
</beans>
```

6.4. Test final de la aplicación completa

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

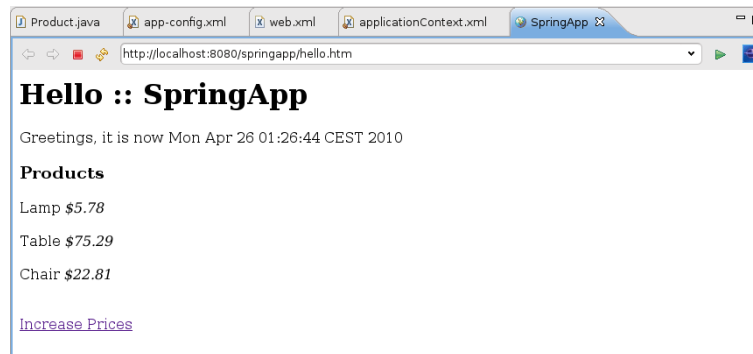


Ilustración 1 - La aplicación completa

La aplicación aparece exactamente como lo hacía antes. Sin embargo, hemos añadido la persistencia en base de datos, por lo que si cierras la aplicación tus incrementos de precio no se perderán sino que estarán todavía allí cuando vuelvas a cargar la aplicación.

6.5. Resumen

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

- Primero hemos modificado la capa de servicio para usar la interface ProductDAO.
- Después hemos tenido que arreglar algunos fallos en los tests de la capa de servicio y la capa web.
- A continuación, hemos introducido un nuevo applicationContext para separar la configuración de la capa de servicio y de la capa de persistencia de la configuración de la capa web.
- Finalmente hemos desplegado la aplicación y testeado que aún funciona.

A continuación puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.

