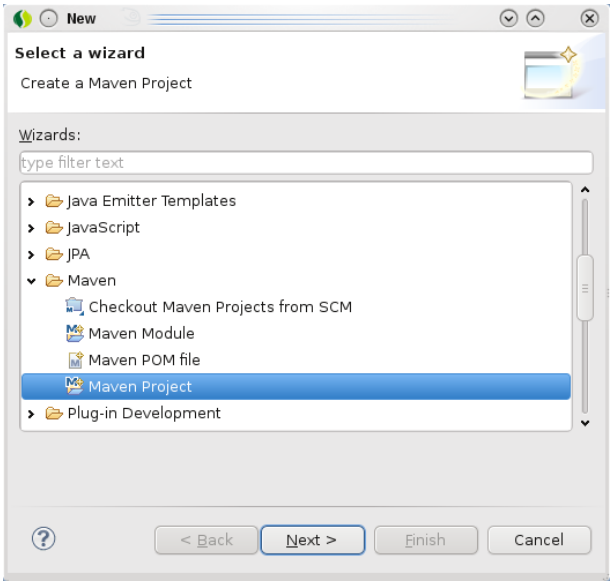
Capítulo 1. Aplicación Base y Configuración del Entorno

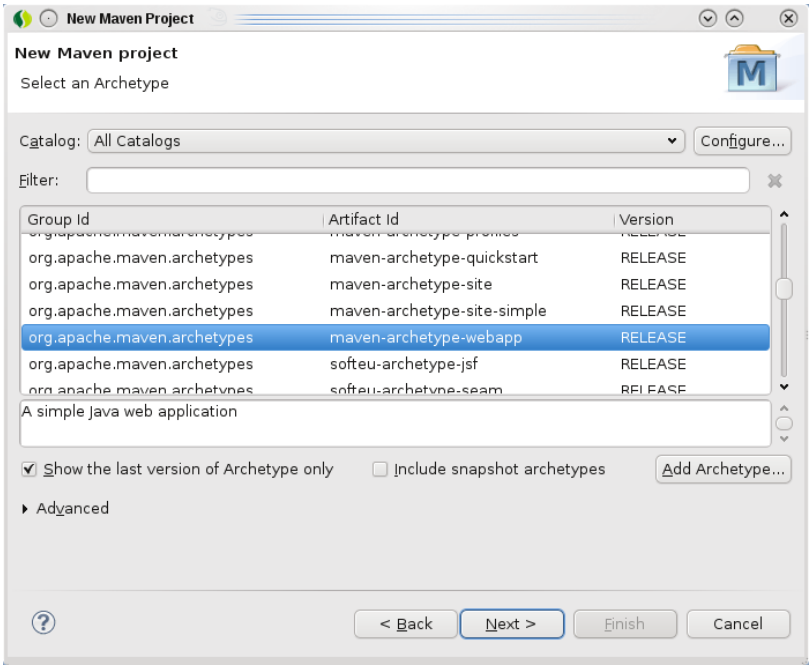
1.1. Crear la estructura de directorios del proyecto

Necesitamos crear un directorio donde alojar todos los archivos que vayamos creando, así que comenzaremos creando un proyecto en el Workspace llamado '**springapp**'. Utilizaremos Maven2 para facilitar la creación de la estructura de directorios del proyecto así como la inclusión de las librerías necesarias.



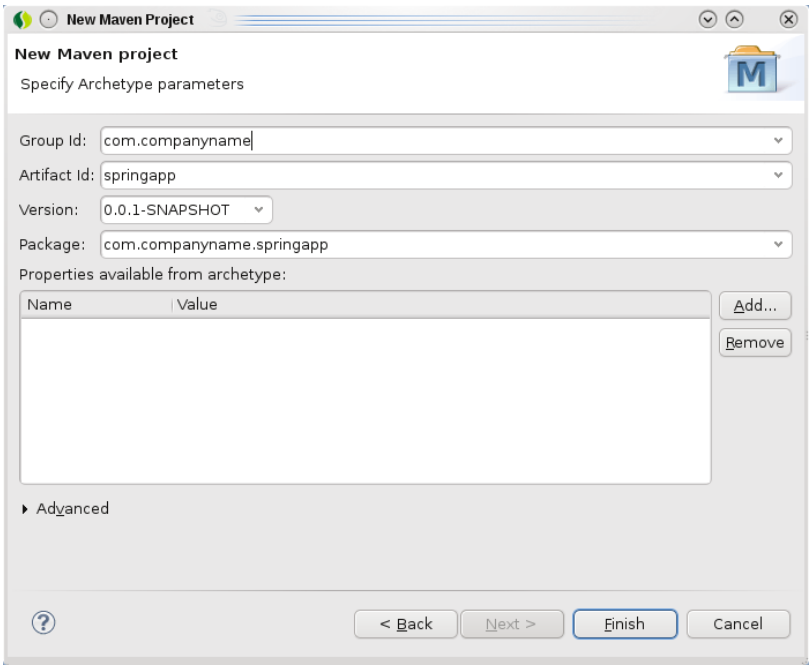
Creación del nuevo proyecto usando Maven

Tras seleccionar el proyecto de tipo Maven y pinchar '`Next`' para mantener la localización del proyecto dentro del Workspace, debemos seleccionar el arquetipo del proyecto. Los arquetipos son patrones o modelos genéricos a partir de los cuales se pueden crear proyectos de un determinado tipo. Maven proporciona un conjunto de estructuras de proyectos (esto es, el árbol de directorios, ficheros que aparecerán en el proyecto, etc.) entre los cuales podemos elegir. De acuerdo con la naturaleza del proyecto que se creará en este tutorial, debemos seleccionar el arquetipo '`maven-archetype-webapp`'.



Selección del arquetipo del proyecto

Maven, entre otras funcionalidades, es útil como herramienta para la gestión de dependencias entre proyectos y permite definir diferentes componentes (llamados artifacts) que forman parte de un grupo de trabajo con una entidad mayor. De momento, sólo disponemos de nuestra simple aplicación para la gestión de inventario.



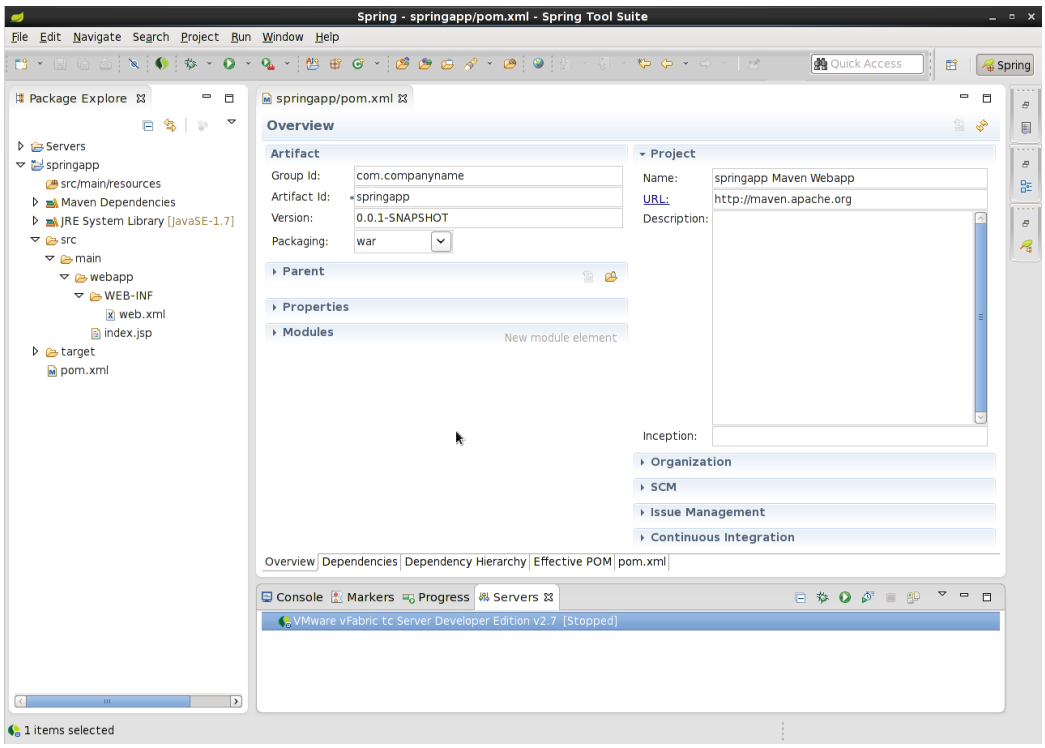
Definición del nombre del proyecto

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.

Si aparecera un warning en el proyecto causado por el uso de diferentes versiones de la JDK, ajustarlas conveniente configurando el **Build Path** del proyecto.

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): no se necesita usar STS para completar este tutorial, pero usándolo podrás hacerlo de manera mucho más sencilla).*



La estructura de directorios del proyecto

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:

'**springapp/src/main/webapp/index.jsp**':

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

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.

'**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"
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

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

</web-app>
```

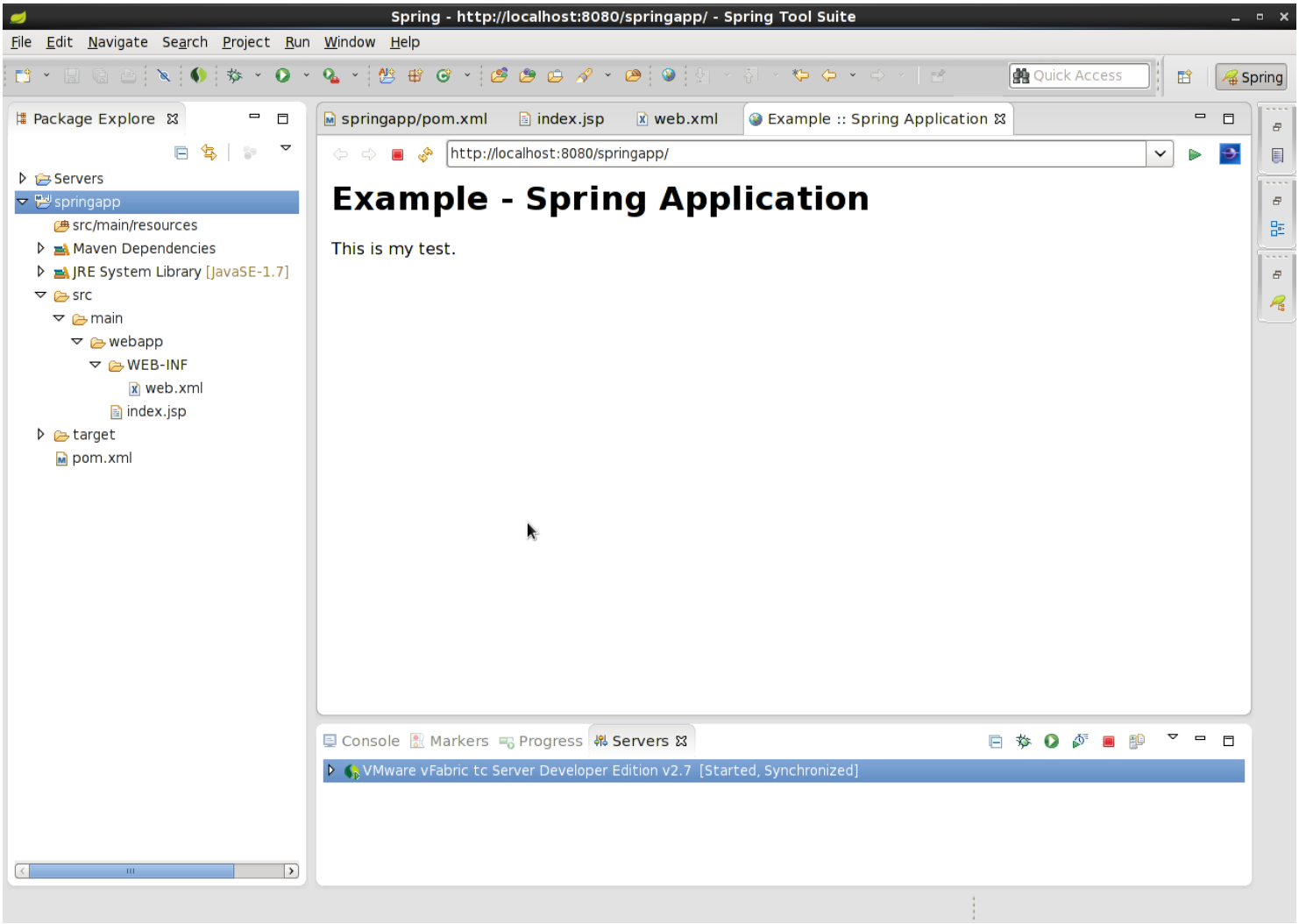
1.3. Desplegar la aplicación en el servidor

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. Por ejemplo: SpringSource tc Server, Tomcat, GlassFish, etc.

1.4. Comprobar que la aplicación funciona

Los pasos anteriores abrirán una pestaña en el entorno de desarrollo STS donde se puede ver el contenido de la página '**index.jsp**'. De manera alternativa, se puede abrir un navegador y acceder a la página de inicio de la aplicación en la siguiente URL: http://localhost:8080/springapp.

(La imagen muestra la visualización sobre STS: el número de puerto puede variar dependiendo del servidor utilizado).



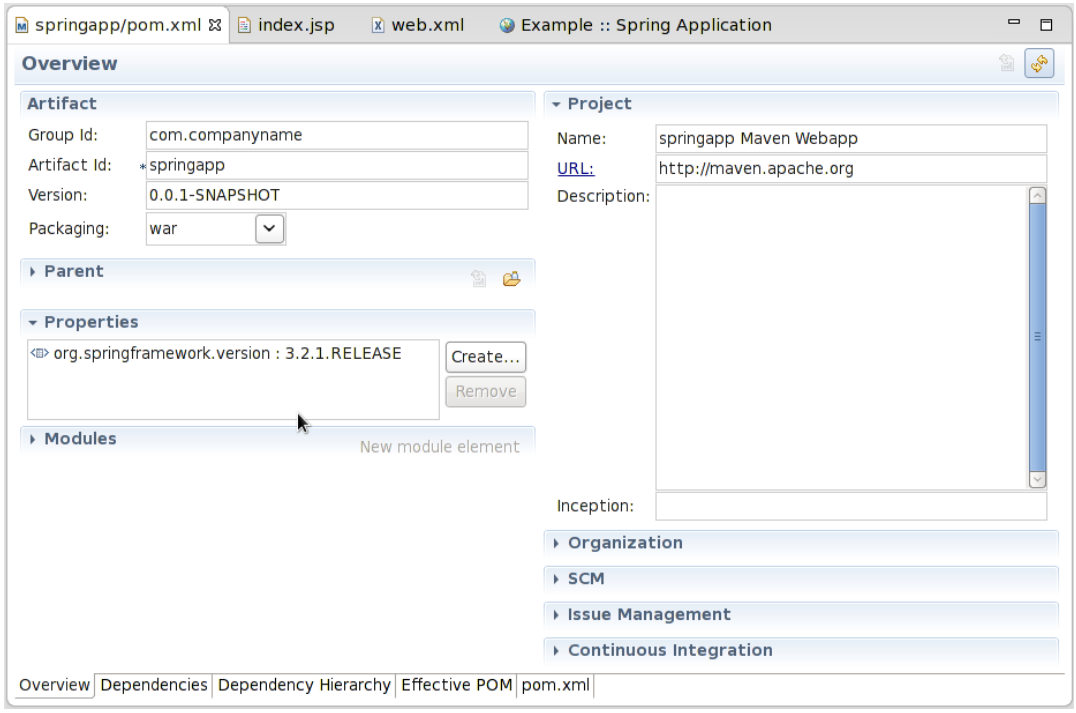
La página de inicio de la aplicación

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 el repositorio Sonatype (http://repository.sonatype.org/index.html) 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 '3.2.0.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 búqueda 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.

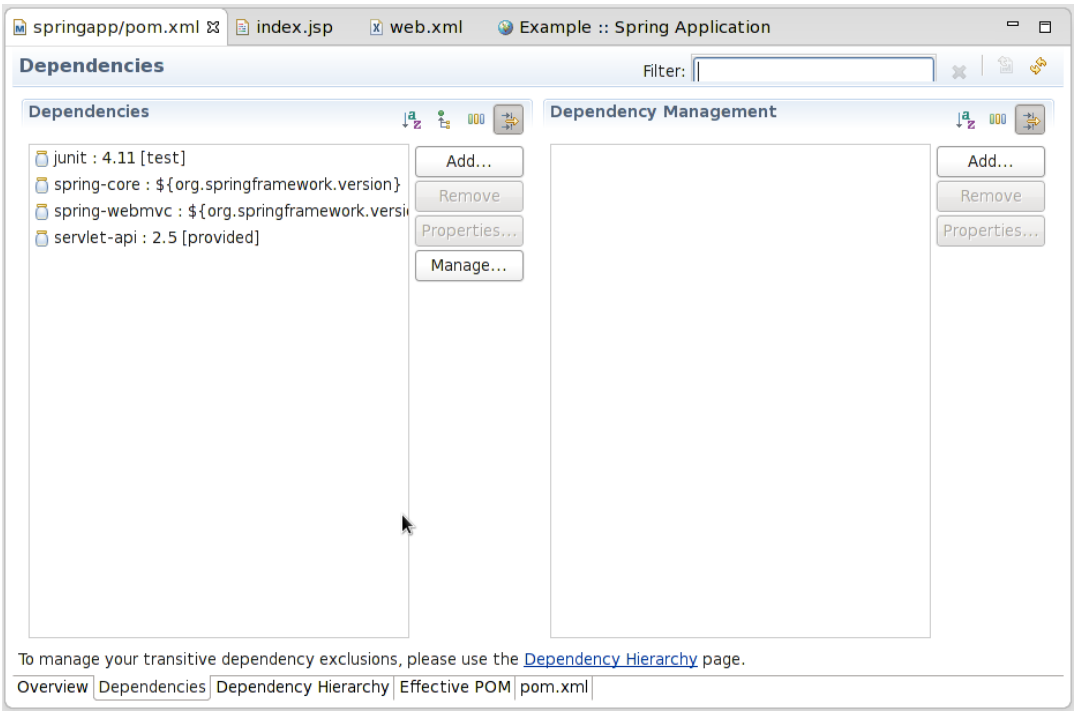


Propiedades del fichero 'pom.xml'

A continuación, crearemos las siguientes dependencias para el proyecto desde la pestaña 'Dependencies' del fichero 'pom.xml'. Las librerías seran 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	4.11	Test
org.springframework	spring-core	${org.springframework.version}	
org.springframework	spring-webmvc	${org.springframework.version}	
javax.servlet	servlet-api	2.5	Provided

Nótese como la dependencia 'junit' (incluida por defecto en el fichero 'pom.xml') ha sido actualizada a la versión '4.11'. 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.



Dependencias del fichero 'pom.xml'

Según el reporte de Roberto Rodríguez, si se está usando WebLogic 11G, es necesario añadir la dependencia de 'joda-time' para evitar un error del tipo 'java.lang.ClassNotFoundException: org.joda.time.LocalDate'. Los detalles de esta dependencia se muestran a continuación.

Group Id	Artifact Id	Version	Scope
joda-time	joda-time	1.6	

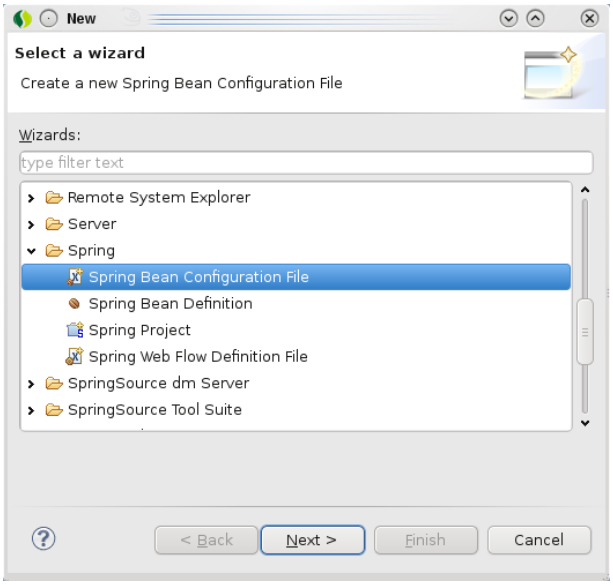
1.6. Modificar 'web.xml' en el directorio 'src/main/webapp/WEB-INF'

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).

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

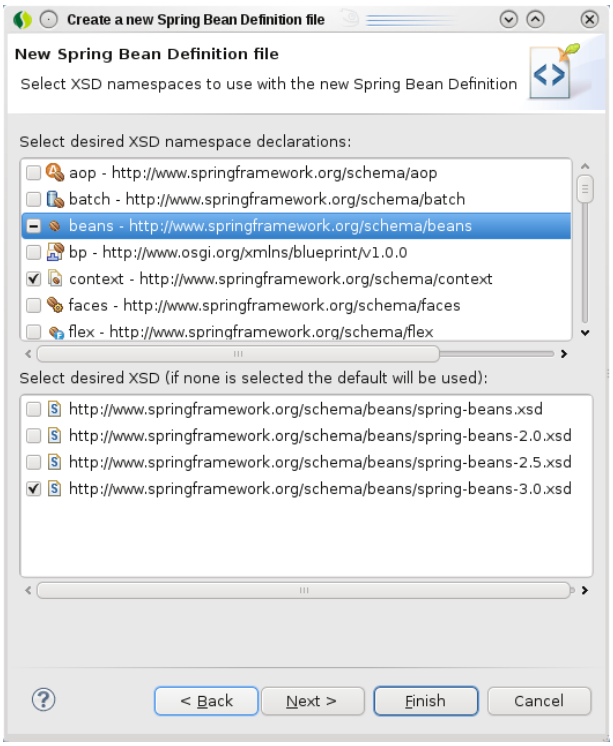
```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:web="http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">

  <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>
```

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 Configuration 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).*



Creación del fichero '**app-config.xml**'

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 '3.2' de Spring Framework.



Selección de los namespaces del 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.

'**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:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="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
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

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

</beans>
```

1.7. Crear el `Controlador`

Vamos a crear una clase anotada con la etiqueta `@Controller` (a la que llamaremos `HelloController`) y que estará definida dentro del paquete '`com.companyname.springapp.web`'. Primero, creamos el directorio '**src/main/java**' como un directorio de tipo '`Source Folder`' donde alojaremos todos los ficheros Java de nuestra aplicación. Dentro de este directorio, creamos el paquete '`com.companyname.springapp.web`'. A continuación, creamos el archivo '**HelloController.java**' y lo situamos en el recién creado paquete. El uso de anotaciones en este fichero requerirá aumentar el nivel de compatibilidad como mínimo a Java 1.5.

'**springapp/src/main/java/com/companyname/springapp/web/HelloController.java**':

```
package com.companyname.springapp.web;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HelloController {

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

    @RequestMapping(value="/hello.htm")
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        logger.info("Returning hello view");

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

Esta implementación del controlador, mediante la anotación `@Controller`, es muy básica. Más tarde la iremos expandiendo. En Spring Web MVC, los componentes `@Controller` *manejan* las solicitudes y devuelven normalmente un objeto `ModelAndView`. En este caso, uno llamado **`'hello.jsp'`**, el cual hace referencia al nombre del archivo JSP que vamos a crear a continuación. El modelo que esta clase devuelve es resuelto a través del `ViewResolver`. Puesto que no hemos definido explicítamente un `ViewResolver`, vamos a obtener uno por defecto de Spring que simplemente redigirá a una dirección URL que coincida con el nombre de la vista especificada. Más tarde modificaremos este comportamiento. Además, hemos especificado un logger de manera que podemos verificar que pasamos por el manejador en cada momento. Usando STS, estos mensajes de log deben mostrarse en la pestaña `'Console'`.

1.8. Escribir un test para el `Controlador`

Los tests son una parte vital del desarrollo del software. El mejor momento para escribir los tests es durante el desarrollo, no después, de manera que aunque nuestro controlador no contiene lógica demasiado compleja vamos a escribir un test para probarlo. Esto nos permitirá hacer cambios en el futuro con total seguridad. Vamos a crear un nuevo directorio de tipo `'Source Folder'` llamado **`'src/test/java'`**. Aquí es donde alojaremos todos nuestros tests, en una estructura de paquetes que será idéntica a la estructura de paquetes que tenemos en **`'src/main/java'`**.

Creamos una clase de test llamada **`'HelloControllerTests'`** dentro del paquete `'com.companyname.springapp.web'`. Para ello seleccionamos el tipo `'JUnit Test Case'` del asistente de creació de ficheros.

`'springapp/src/test/java/com/companyname/springapp/web/HelloControllerTests.java'`:

```
package com.companyname.springapp.web;

import static org.junit.Assert.*;

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


public class HelloControllerTests {

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

}
```

Para ejecutar el test (y todos los tests que escribamos en el futuro), basta con que seleccionemos `'Run as > JUnit Test'` sobre el menú contextual que aparece cuando se pincha el botón derecho sobre la clase de test. Si el test se ejecuta de forma satisfactoria podrás ver una barra verde que lo indica.

1.9. Crear la `Vista`

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

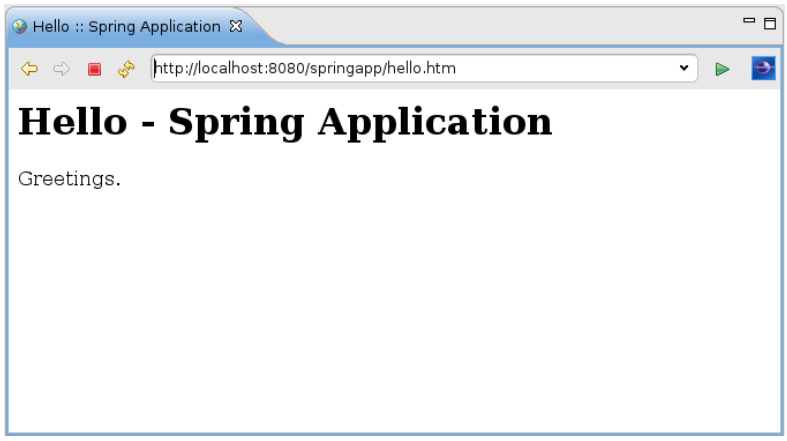
`'springapp/src/main/webapp/hello.jsp'`:

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

1.10. Compilar, desplegar y probar la aplicación

Para ejecutar la aplicación de nuevo, seleccionamos de nuevo `'Run as > Run on Server'` sobre el menú contextual que aparece cuando se pincha el botón derecho sobre el nombre proyecto. *(En algunos casos, es necesario reiniciar el servidor para asegurar que la aplicación se actualiza correctamente).*

Probemos esta nueva versión de la aplicación. Desde el navegador que ofrece STS, o desde cualquier otro navegador, abrir la URL http://localhost:8080/springapp/hello.htm.



La aplicación actualizada

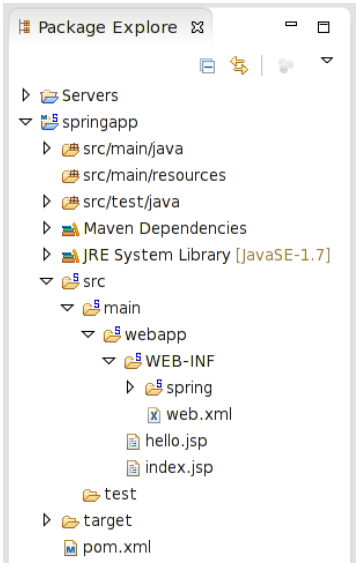
1.11. Resumen

Echemos un vistazo rápido a las partes de nuestra aplicacioón 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 enlance a nuestra aplicación.
- Un controlador frontal, `DispatcherServlet`, con el correspondiente archivo de configuración **`'app-config.xml'`**.
- Un controlador de página, `HelloController`, con funcionalidad limitada – simplemente devuelve un objeto `ModelAndView`. Actualmente tenemos un modelo vacío, más tarde proveeremos un modelo completo.
- Una unidad de test para la página del controlador, `HelloControllerTests`, para verificar que el nombre de la vista es el que esperamos.

- Una vista, `'hello.jsp'`, que de nuevo es extremadamente sencilla. Las buenas noticias son que el conjunto de la 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.



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