Sistemas Distribuidos e Internet
Arquitectura MVC con Spring Boot
Sesión 2
Curso 2019/2020

1 Introducción al desarrollo de aplicaciones Web basadas en MVC con Spring Boot

En esta práctica veremos una introducción de como desarrollar aplicaciones Web basadas en el patrón MVC con Sprint Boot.

1.1 Configuración del desarrollo

La forma más común de desarrollar aplicaciones web utilizando el framework Spring es utilizando el **Spring Tool Suite (STS)**. https://spring.io/tools Este entorno está basado en el IDE eclipse e incluye todo lo necesario para facilitarnos el desarrollo de aplicaciones Spring.

En el caso de no tener STS instalado en el equipo debemos descargar la versión del STS correspondiente a nuestro sistema operativo y lo descomprimimos (Requiere que tengamos instalado Java 1.8 o superior). Además, vamos a integrar nuestro proyecto en un repositorio Git.

Nota: Inicialmente vamos a crear el proyecto inicial en STS y luego lo sincronizaremos con un repositorio remoto en GitHub.

1.2 Spring

Spring es uno de los frameworks más populares para el desarrollo de aplicaciones Web sobre la plataforma Java.

Spring cuenta con una gran cantidad de módulos que proveen servicios de diferentes tipos: patrón arquitectónico MVC (Modelo Vista Controlador), contenedor de inversión de control[1], inversión de dependencias[2] e inyección de dependencias[3], acceso a datos, gestión de transacciones, acceso remoto, mensajería, administración remota, autentificación y autorización, etc.

Spring Boot

1 El framework invoca al código ante ciertos eventos, tales como pulsar un botón, etc.

2 Según Martin Fawler, las clases de las capas superiores no deberían depender de las clases de las capas inferiores, sino que deberían basarse en abstracciones (patrón N-Capas de Brown).

3 Este concepto se basa en hacer que una clase A inyecte objetos en una clase B en lugar de dejar que sea la propia clase B la que se encargue de crear el objeto.

https://projects.spring.io/spring-boot/ Es un proyecto Spring que utiliza las características del framework combinadas con algunas nuevas creadas específicamente para conseguir desarrollos y despliegues **más agiles**.

- Crea "Stand-alone Spring Applications", que se ejecutan en un servidor Tomcat, Jetty o Undertow embebido, sin necesidad de desplegar WARs.
- Provee un fichero POM simplificado para configurar las dependencias de librerías del proyecto en MAVEN.
- Realiza múltiples configuraciones de Spring de forma automática, cuando estas son posibles.
- Provee varias características y métricas para aplicaciones en producción y la posibilidad de externalizar muchas configuraciones.
- No requiere configuración vía XML.

1.3 Creación de proyecto y despliegue

Un proyecto Spring Boot es básicamente un proyecto Java Maven o Gradle, con al menos la dependencia a la librería "spring-boot-starter-web". También se puede utilizar los lenguajes de programación Kotlin y Groovy que se ejecutan sobre la máquina Virtual de Java.Las tres alternativas más habituales para crear un nuevo proyecto Spring boot son:

1. Crear un proyecto Maven desde el entorno e incluir la dependencia en el pom.xml (maven)
2. Generar una plantilla a través del inicializador de Spring: https://start.spring.io/ . Seleccionando al menos la dependencia "Web" (Full-stack web development with Tomcat and Spring MVC)
3. Crear un proyecto "Spring Starter Project" e incluir la dependencia web desde el asistente.

Vamos a optar por la opción 3 ya que es la más directa, aunque las otras serian igualmente válidas.

¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

Aunque en este guión se ha usado como nombre de repositorio para todo el ejercicio **Notaneitor**, cada alumno deberá usar como nombre de repositorio **sdix-lab-spring**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI1920.pdf del CV**.**

Por ejemplo el alumno con IDGIT=101, deberá crear un repositorio con nombre sdi1920-101-lab-spring y un proyecto SpringBoot en STS con nombre también sdi1920-101-lab-spring.

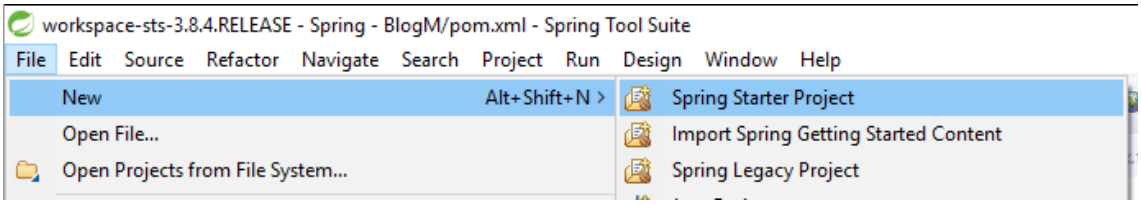
En resumen:

Nombre repositorio GitHub: **sdix-lab-spring**

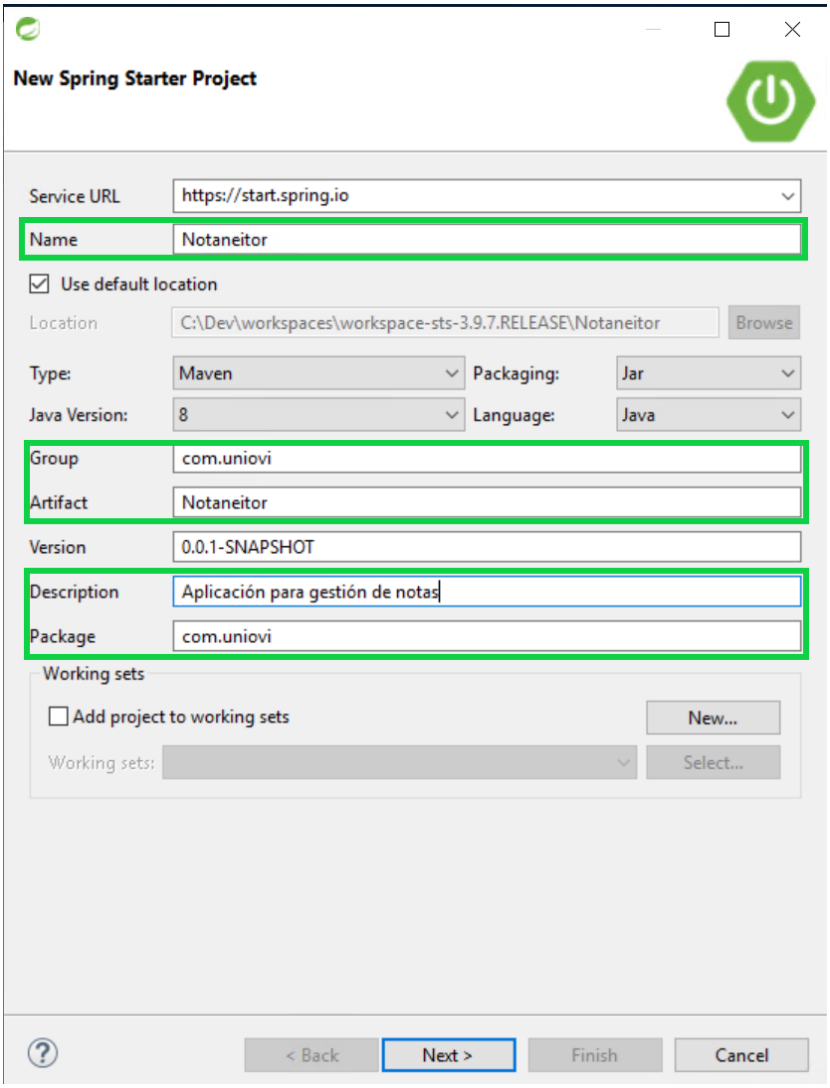
Nombre proyecto STS : **sdix-lab-spring**

Otra cuestión **IMPRESCINDIBLE** es que una vez creado el repositorio deberá invitarse como colaborador a dicho repositorio a la cuenta github denominada **sdigithubuniovi.**

File -> New -> Spring Starter Project

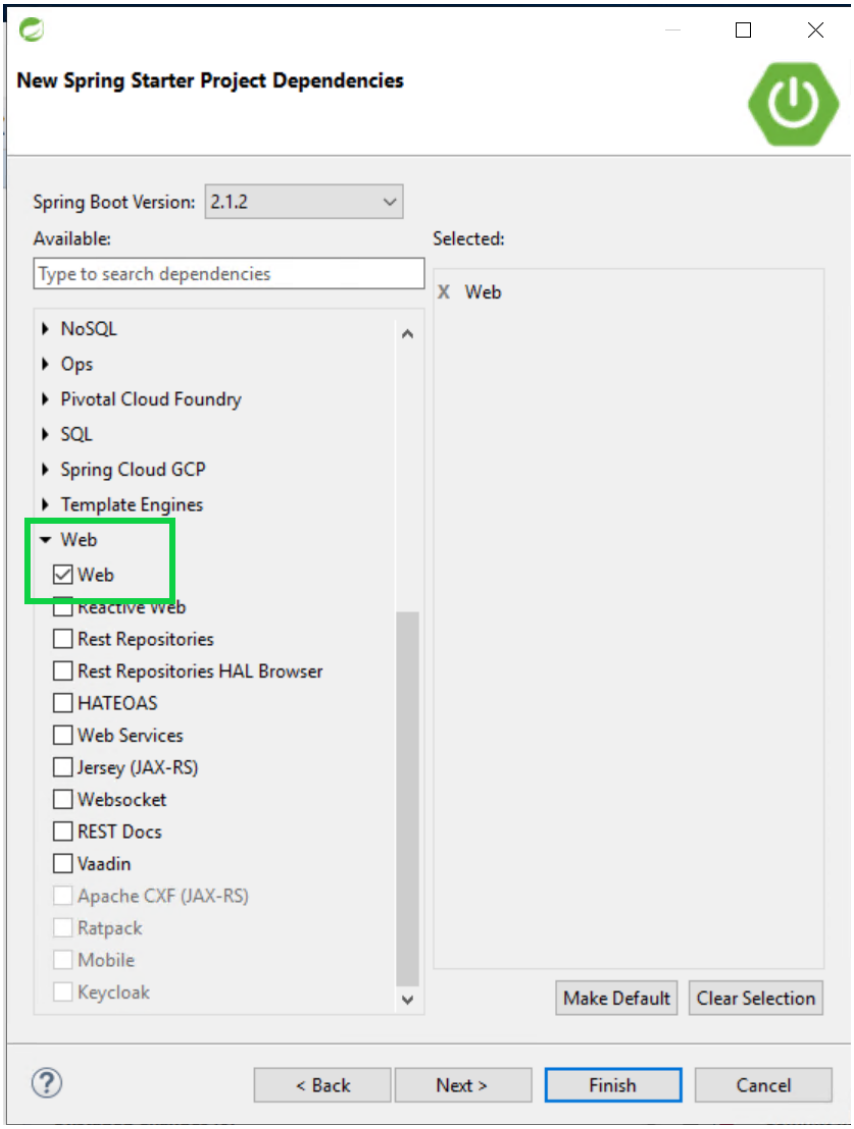


Le damos el nombre "Notaneitor" al proyecto. Nos aseguramos de cambiar todos los campos en el asistente para que coincidan con los de la siguiente captura:



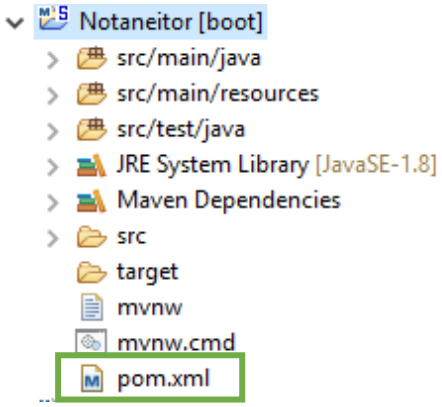
Pulsamos en **Next** y seleccionamos la **dependencia Web -> Web**[4] en el asistente. Estas son las dependencias con las que se iniciará el proyecto, pero en el futuro podemos añadir más si las necesitamos.

[4] Para STS en Sistemas MACS seleccionar en el desplegable Spring Boot Version 2.1.2, suponiendo que se está trabajando con STS 3.9.7.



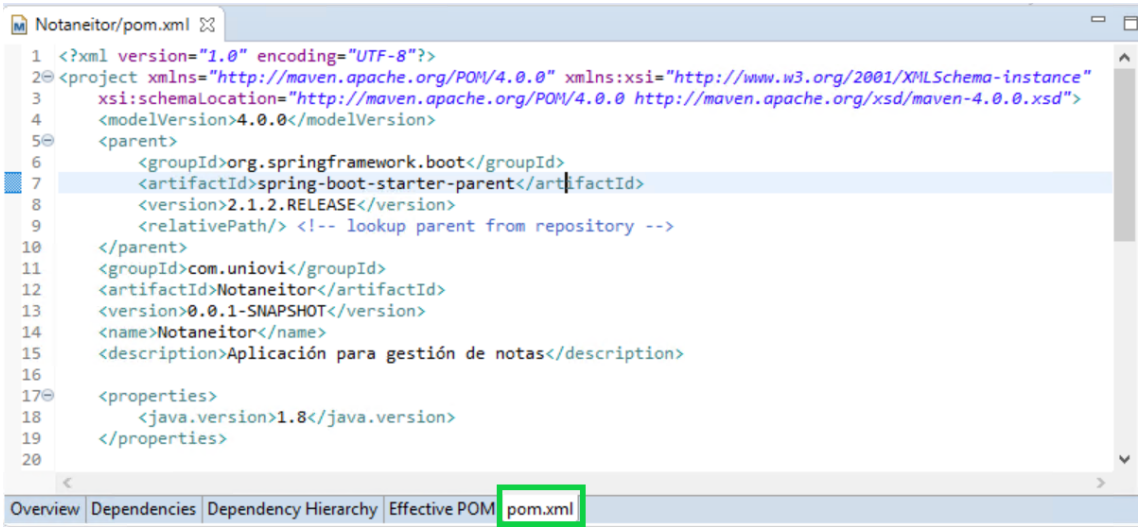
Pulsamos en **Finish** y con esto finalizamos la creación de la plantilla del proyecto. El proyecto en sí tiene un contenido minimo, basicamente una estructura de carpetas, una clase principal y varios ficheros de configuración.

Nota: Una vez se descargen todas las dependencias del proyecto, veremos la estructura de carpetas tal como se muestra en la siguiente imagen.



Uno de los ficheros más interesantes es el **pom.xml** (Maven gestión de librerias del proyecto).

Seleccionamos el fichero **pom.xml** y cambiamos la vista del asistente a la vista XML utilizando las pestañas inferiores.



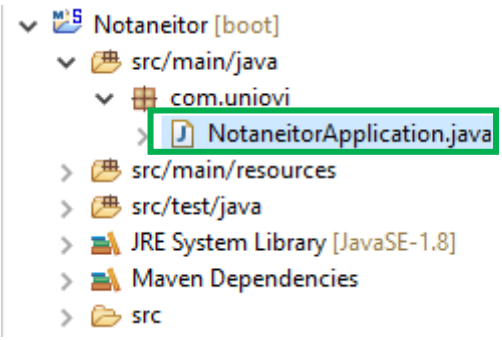
Aquí se definen varias aspectos de la aplicación entre los que conviene destacar:

- Las propiedades de la aplicación: groupId, artifactId, version, packaging, name,description.
- Parent : dependencia principal: org.springframework.boot
- Propiedades del proyecto: codificación y versión de Java
- Grupo de dependendencias: actualmente la dependencia **spring-boot-starter-web** y **spring-boot-starter-test** .

De momento no vamos a modificar ninguno, pero casi todas las futuras modificaciones en el Maven se van a centrar en agregar elementos al bloque <dependencies>. Esta es la forma de agregar librerias a un proyecto utilizando Maven, al incluir la librería en la sección <dependencies> esta se agrega al proyecto automaticamente.

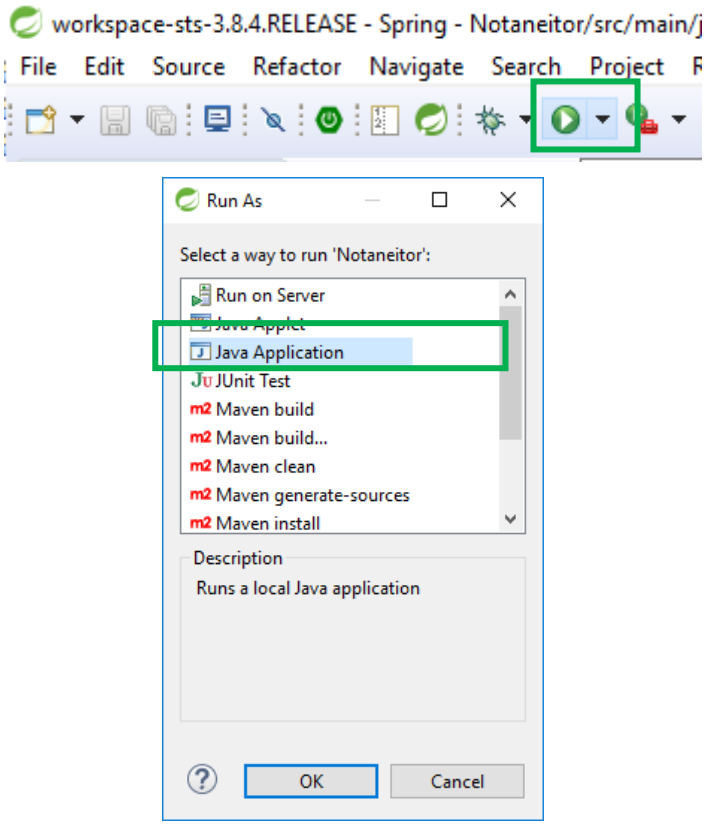
```
<dependencies>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>

	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-test</artifactId>
		<scope>test</scope>
	</dependency>

</dependencies>
```

Dentro de la carpeta src/main/java/ tenemos la clase principal del proyecto: **NotaneitorApplication.java**. Se trata de una clase java normal, con la anotación **@SpringBootApplication** y un método main.

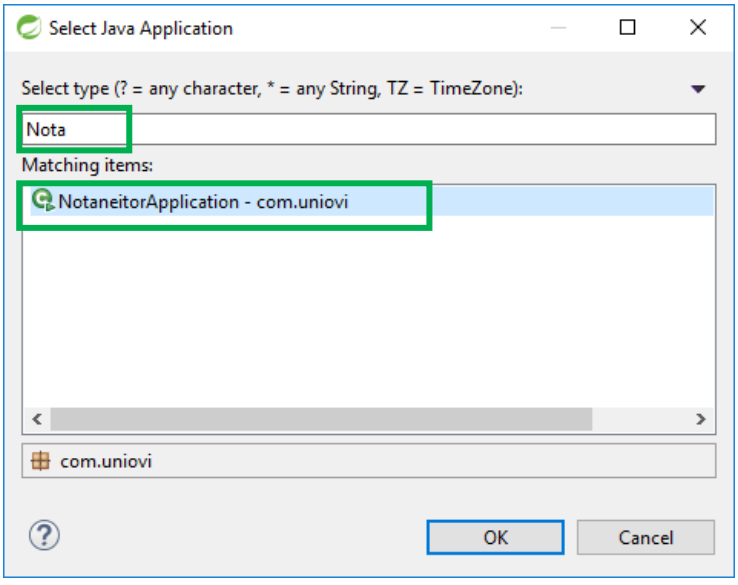



```
@SpringBootApplication
public class NotaneitorApplication {

    public static void main(String[] args) {
        SpringApplication.run(NotaneitorApplication.class, args);
    }
}
```

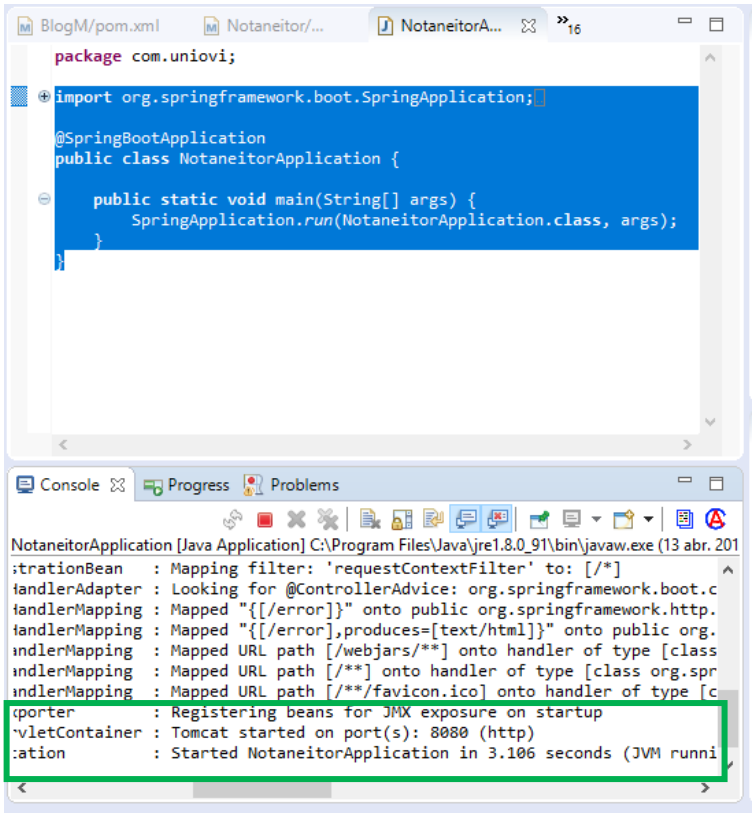
Esta es la clase principal de la aplicación, si marcamos la clase en el proyecto como una aplicación Java (**Java Application**) al ejecutar el proyecto con la opción "Run/Play" (triángulo blanco) de la barra superior.



Cuando nos pregunte por la clase principal seleccionamos la clase que tiene el método Main: **NotatenitorApplication**.

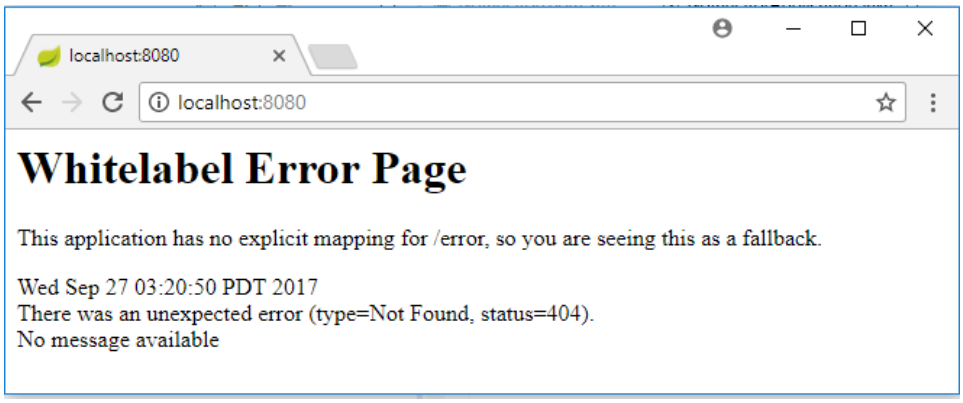


Veremos al ejecutar la aplicación se empieza a mostrar información en la ventana **Console** del eclipse. Las aplicaciones Spring Boot son "Standalone applications", al ejecutarlas despliegan su propio servidor.



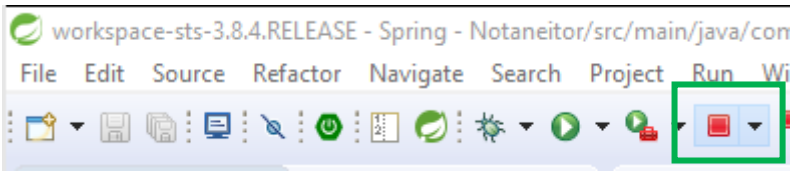
Según la información mostrada en la consola, tenemos un Tomcat en el puerto 8080. Abrimos la URL http://localhost:8080 en el navegador.

Sí se nos muestra la siguiente página significa que todo ha funcionado correctamente. Pero nuestra aplicación aun no está configurada para responder a ninguna petición.

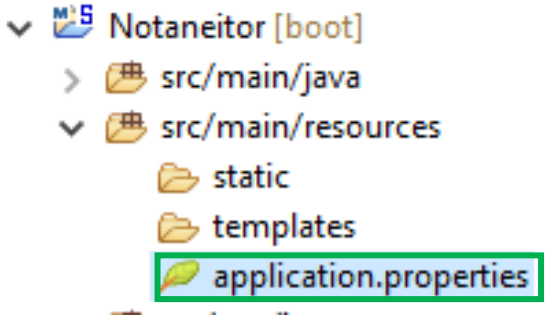


Sí el puerto 8080 del equipo está ocupado por otro proceso lo cambiaremos tal y como se indica a continuación.

Para detener la aplicación pulsamos en el botón **Stop**. Debemos recordar que hay que parar y volver a desplegar la aplicación cada vez que realicemos cambios.



Abrimos el fichero de propiedades situado en **src/main/resources -> application.properties** e incluimos la siguiente propiedad.




```
server.port = 8090
```

Para que los cambios tengan efectos debemos recordar salvar el fichero **application.properties**. y parar/iniciar la aplicación

Observamos que por defecto el fichero esta vacio, eso se debe a que no hace falta especificar ninguna propiedad sino que todas tienen valores por defecto, este framework sigue la filosofia de "Convención sobre configuración ", todas las propiedades usan valores por defecto, unicamente los definimos si queremos modificar el valor establecido por convención.

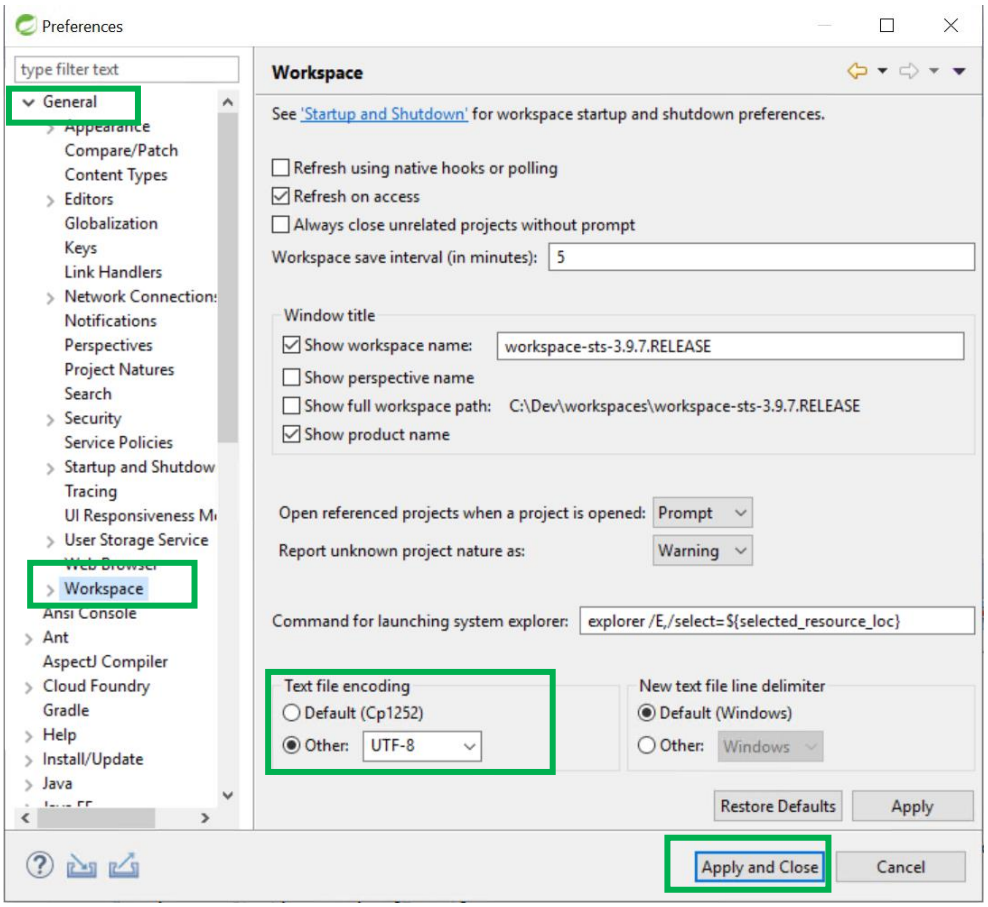
Nota: A partir de este punto en el guión se desplegara la aplicación en el puerto 8090.

1.4 Códificación del workspace UTF-8

Antes de continuar para ahorrarnos futuros problemas de codificación es buena idea cambiar la codificación del eclipse a UTF-8.

Window -> preferences

En la lista de la izquierda seleccionamos **General** > **Workspace**



Despues modificamos la propiedad **Text file encoding**, cambiando el valor a **UTF-8**, al aplicar los cambios nos pedirá reiniciar el Eclipse.[5]

[5] En sistemas MACOS el juego de caracteres por defecto es UTF-8 por lo que no será necesario hacer este cambio.

1.5 Gestión de dependencias con librerías

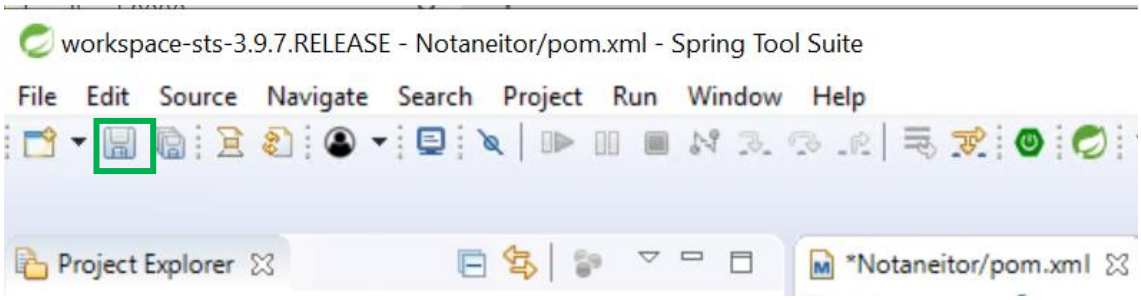
Las librerías externas son uno de los factores claves para desarrollar una aplicación web Spring. Normalmente nos valemos de las librerías externas muchos aspectos importantes: la seguridad, gestión de datos, etc.

Las librerías incluyen muchas funcionalidades extras que puede ser usada dentro de la aplicación. Todas las librerías de nuestra aplicación se gestionan a través de Maven, (no vamos a importar ningún .jar en el proyecto).

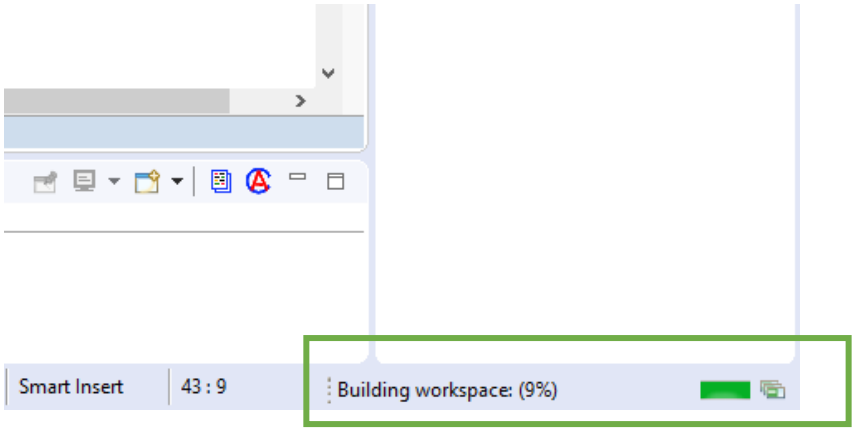
A modo de prueba, vamos a comenzar importando una dependencia, para poder utilizar la librería de JSTL (JSP - Standard Tag Library) en la aplicación. Incluimos la siguiente etiqueta dentro del boque **<dependencies>.**

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>

</dependencies>
```

Al guardar los cambios en el fichero **pom.xml** podemos observar que el eclipse vuelve a construir el Workspace automáticamente (descarga la librería jstl y la agrega al path del proyecto).

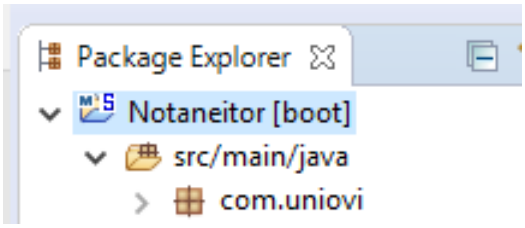


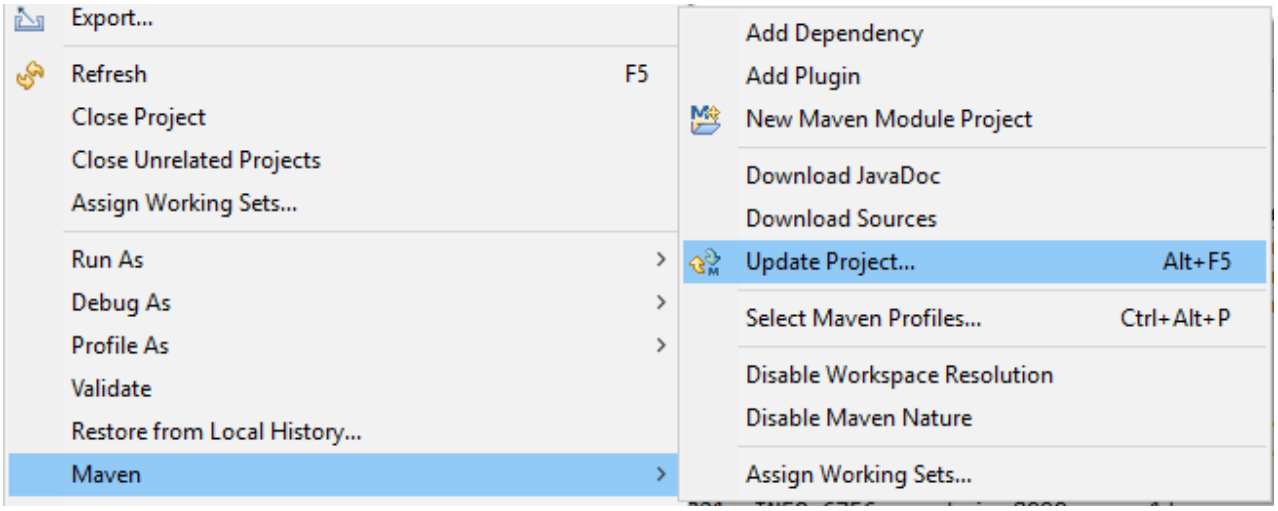
Nota: En este caso la reconstrucción del workspace va a ser muy rápida.



También existe una forma de forzar la actualización de las dependencias del proyecto (aunque en este entorno no suele ser necesario ya que se actualiza automáticamente al salvar).

Click derecho encima del **nombre proyecto -> Maven -> Update Project…**

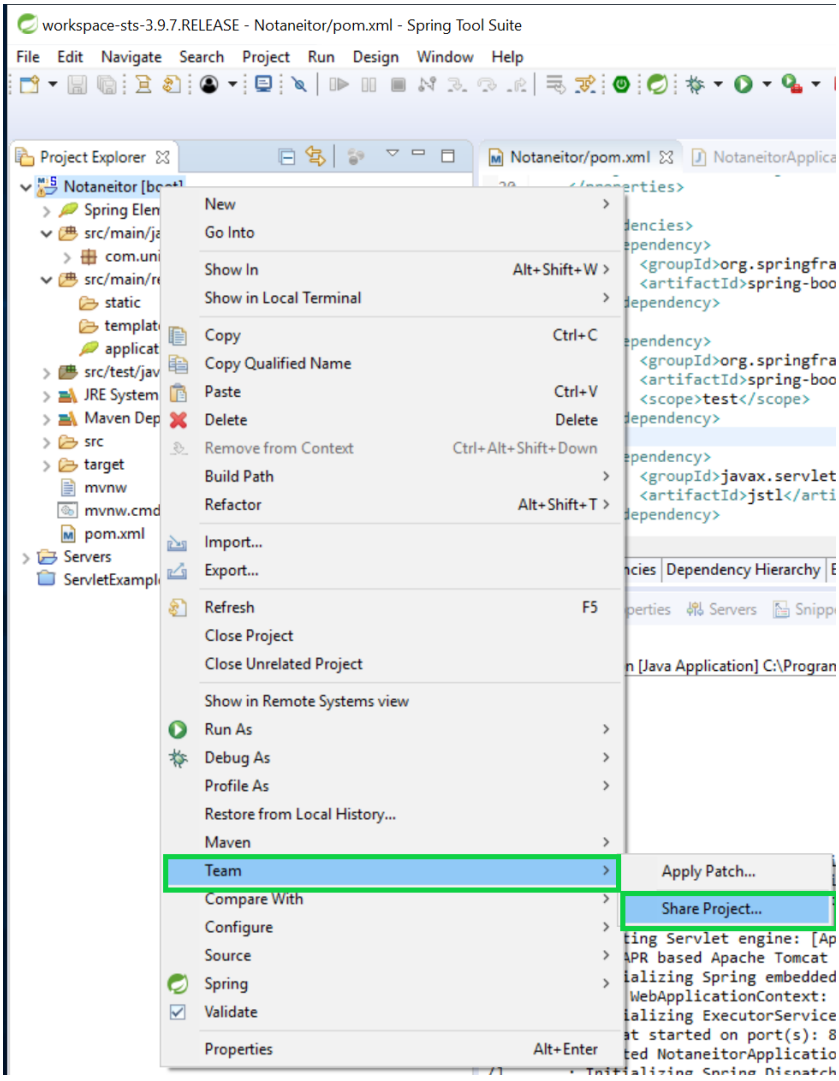




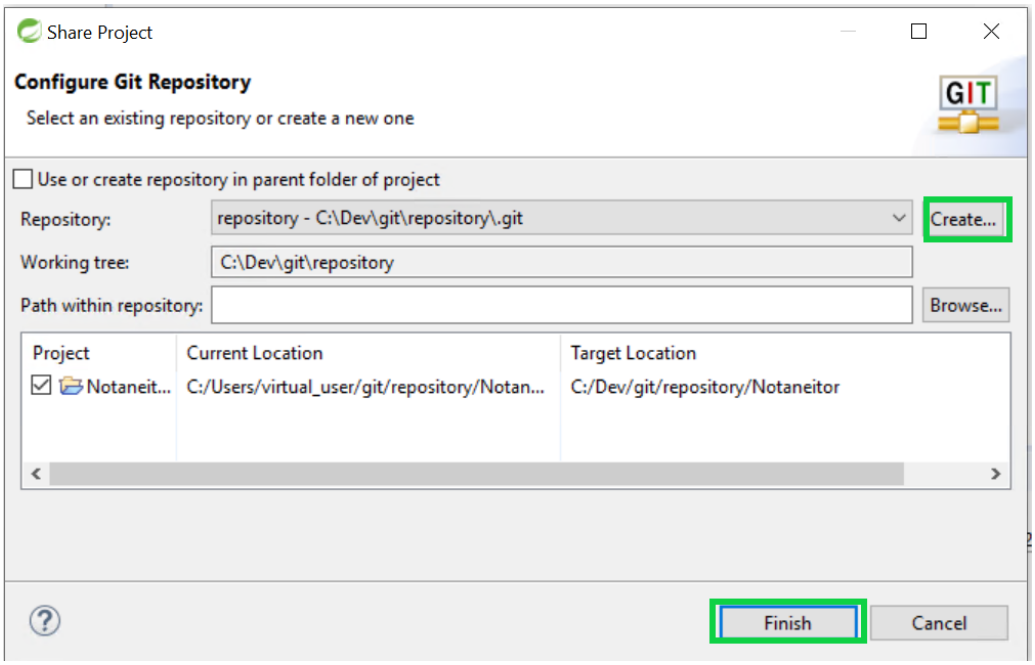
1.6 Configurar el repositorio Git

Actualmente nuestro proyecto no está asociado a **ningún repositorio**, por esto, en esta sección vamos subir el proyecto a GitHub usando Git como software de control de versiones.

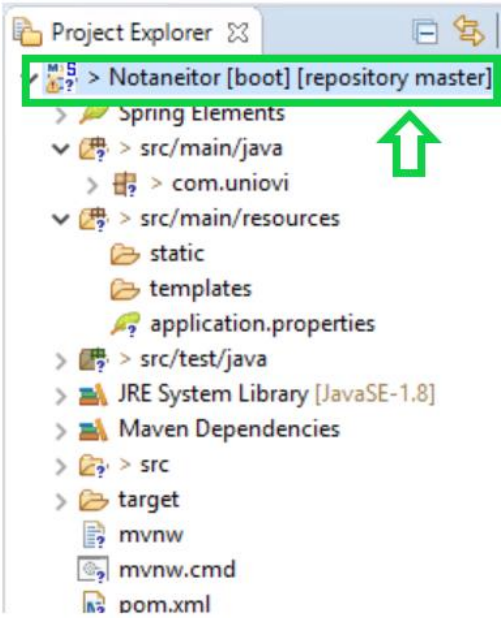
Lo primero que vamos a hacer es hacer click derecho sobre el proyecto e ir a la opción de **Team (Equipo)** -> **Share Project** (compartir proyecto).



A continuación, aparecerá una ventana desde donde podremos usar o crear uno repositorio nuevo. En nuestro caso, haciendo click en el botón **"create"** nos permitirá seleccionar una carpeta local **en donde crear el nuevo repositorio.**

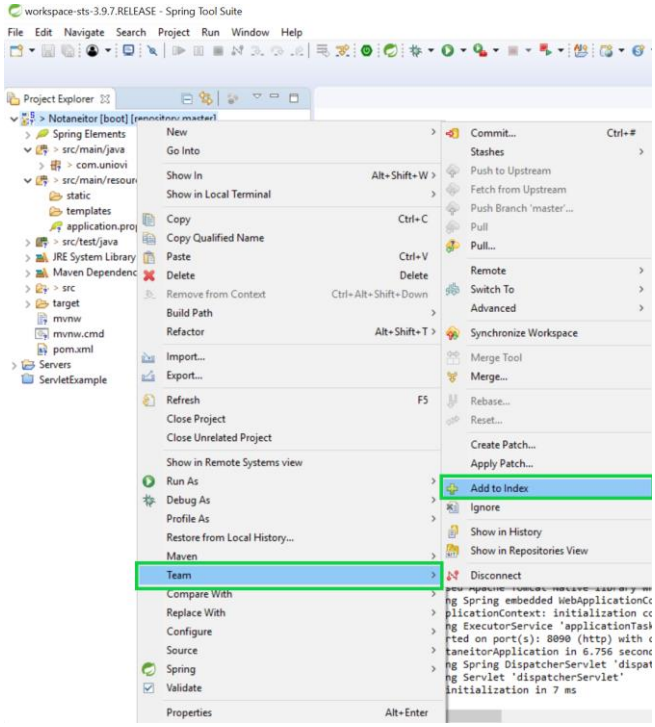


Al hacer click en el botón **Finish** en la ventana nos aparecen los iconos del proyecto en STS como se muestra a continuación.

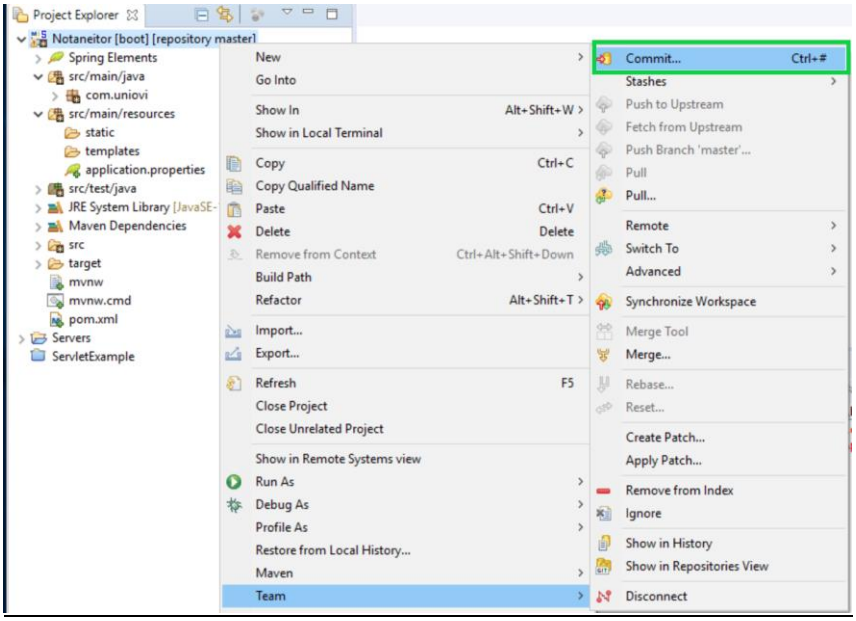


Añadir el proyecto al repositorio local

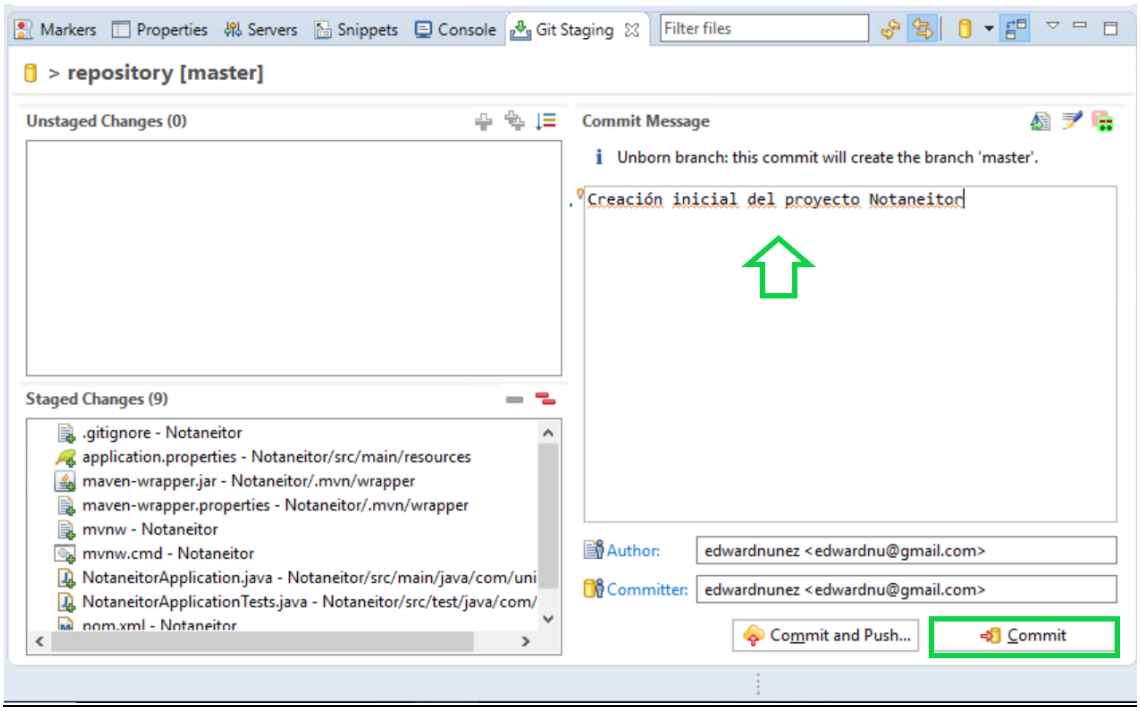
Ahora el proyecto esta listo para ser añadido al repositorio local. Para añadir todos los ficheros del proyecto a GIT vamos a la opción **Add to Index** de menú **Team** de STS.



Ahora hacemos un **Commit** al repositorio local.



En la ventana de Git Staging, escribimos el **mensaje del commit** y hacemos click en el botón **Commit**.



Añadir el proyecto al repositorio remoto (GitHub)

Para añadir el proyecto al repositorio remoto GitHub, creamos un repositorio en GitHub. En este guión lo hemos llamado Notaneitor como nombre genérico, pero hay que recordar siguir la nomenclatura indicada anteriormente usando el nombre sdix-lab-spring.

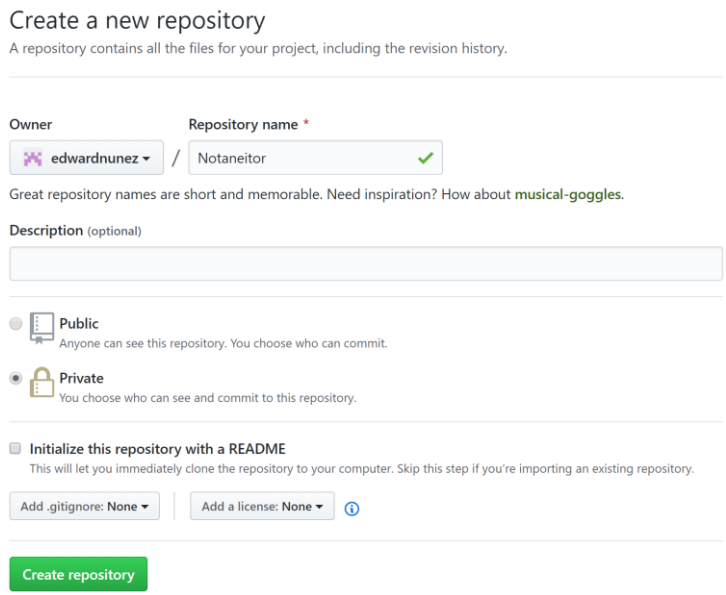
¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

Aunque en este guión se ha usado como nombre de repositorio para todo el ejercicio **Notaneitor**, cada alumno deberá usar como nombre de repositorio **sdix-lab-spring**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI1819.pdf del CV**.**

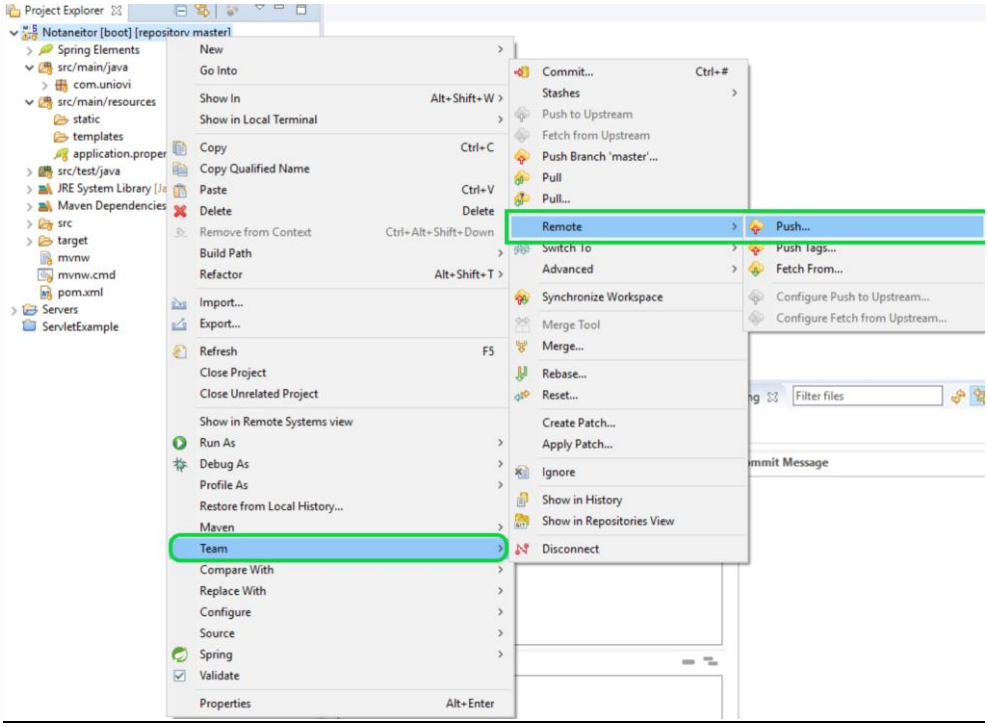
En resumen:

Nombre repositorio GitHub: **sdix-lab-spring**

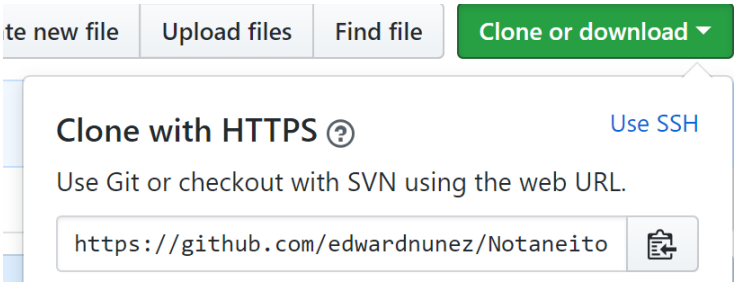
Otra cuestión **IMPRESCINDIBLE** es que una vez creado el repositorio deberá invitarse como colaborador a dicho repositorio a la cuenta github denominada **sdigithubuniovi.**

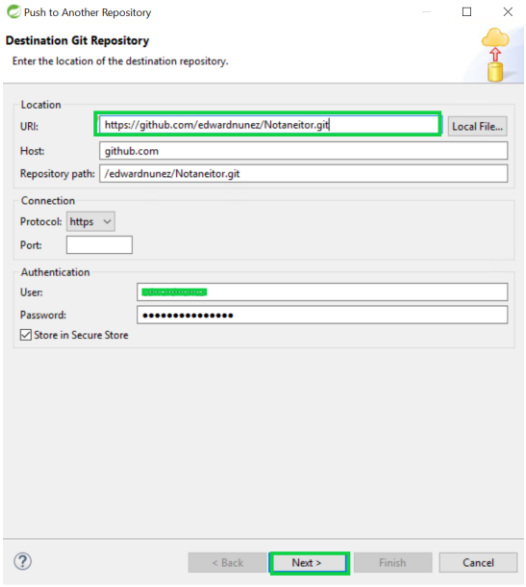


Una vez creado el repositorio de GitHub, lo siguiente es **subir el código al repositorio remoto**, para ello vamos a la opción de STS, **Team->Remote->Push.**

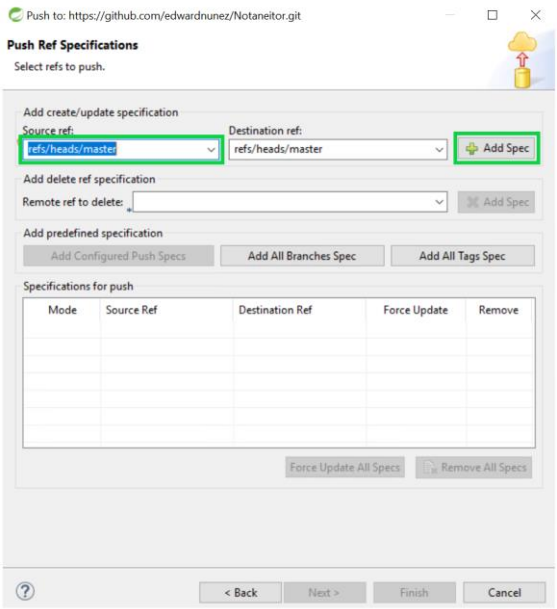


En la nueva ventana especificamos la **URL del repositorio** que hemos creado en GitHub. Solo tenemos copiar la URL del repositorio desde nuestra cuenta de GitHub y **copiarla en el cuadro de texto URL de la ventana, especificar nuestras credenciales** y hacer click en el botón **Next**.

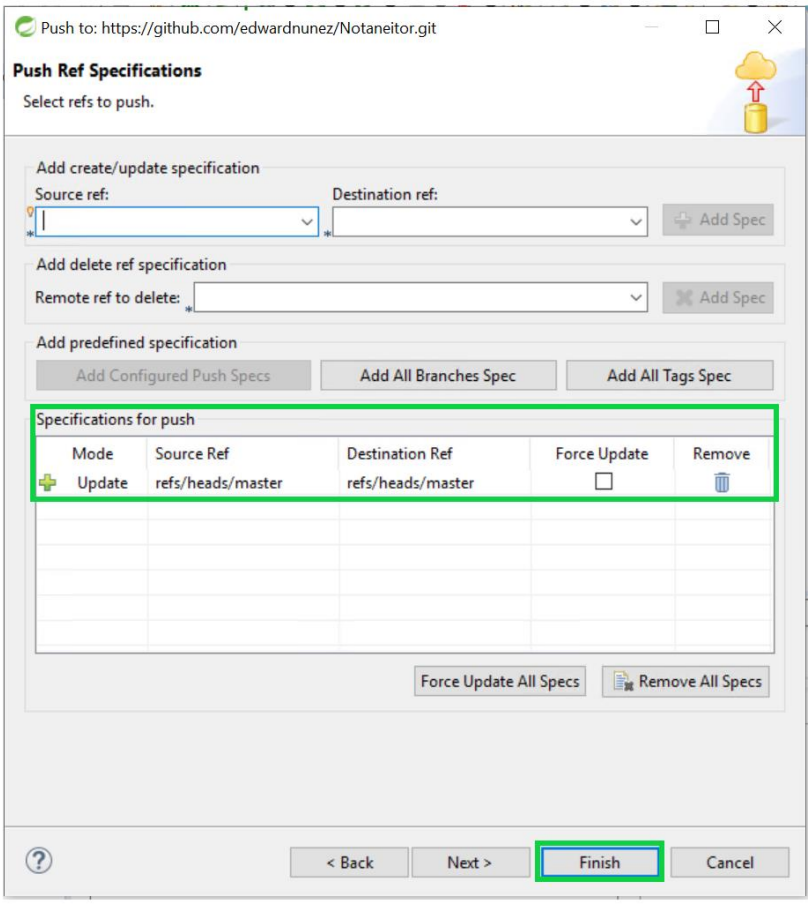




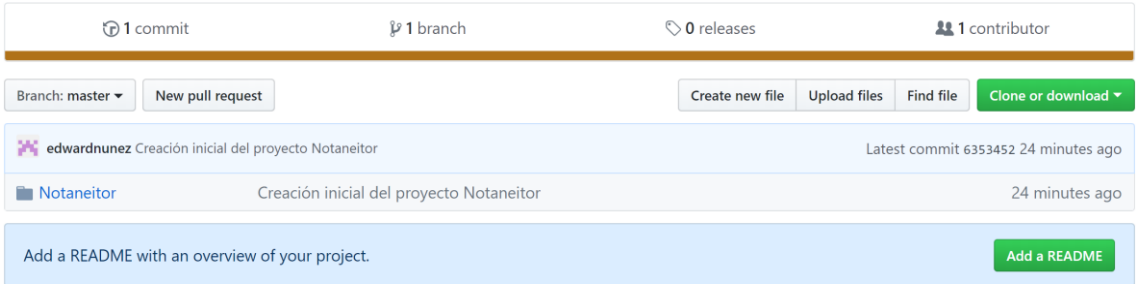
Seleccionamos la opción de **ref/head/master** en Source ref y hacemos click en el botón **Add Spec.**



Una vez añadidas las especificaciones del repositorio hacer click en el botón **Finish** para subir el código al repositorio remoto.



Finalmente, en nuestra cuenta de GitHub podremos ver los ficheros del proyecto.



1.7 Arquitectura MVC

La arquitectura Modelo-Vista-Controlador (MVC) se utiliza comúnmente en el desarrollo de aplicaciones Web, así como en cualquier aplicación que presenta algún tipo de interacción con el usuario. Tiene como objetivo principal conseguir la separación de responsabilidades en la aplicación, con los consiguientes beneficios que esto puede proporcionar en un desarrollo:

- Mejorar la arquitectura y la robustez
- Fuerzan a utilizar una arquitectura modular
- Fomentan la separación entre capas
- Favorece la reutilización
- En general esta arquitectura favorece el manteamiento y la extensión de la aplicación en comparación con arquitecturas basadas en otros patrones.

Esta arquitectura se divide en tres capas claramente diferenciadas: el **Modelo**, la **Vista** y el **Controlador**.

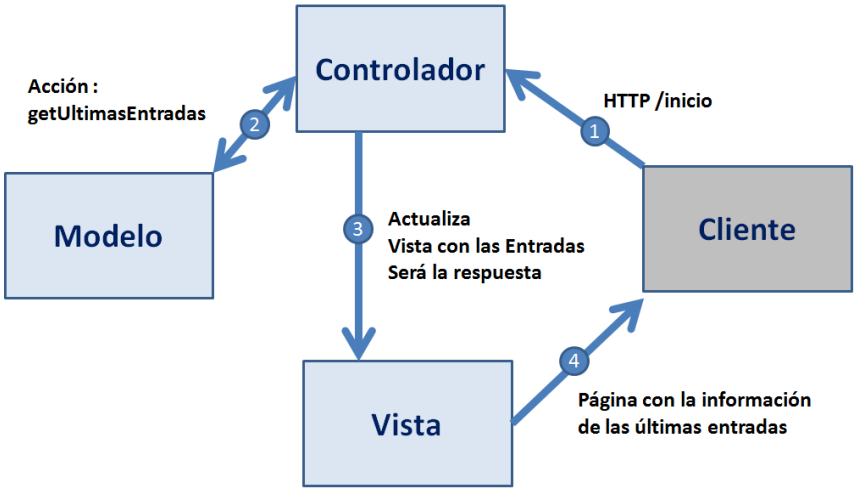
Modelos: definen los mecanismos pertinentes para gestionar los datos de la aplicación. A través de los modelos se accede al sistema de persistencia y se crean, modifican o recuperan los datos. En la implementación de los modelos encontraremos código que se ejecuta en el servidor y que accede y manipula directamente los repositorios de datos de la aplicación.

Vistas: definen el aspecto de interfaz de usuario de la aplicación, es decir la parte que será enviada a los navegadores web de los clientes y por lo tanto la información que los usuarios van a percibir. En la mayor parte de aplicaciones las vistas estarán compuestas por código HTML, CSS y JavaScript, pudiendo incluir también pequeños scripts de lenguaje ejecutado en el servidor, que acostumbra a utilizarse como vínculo de unión entre las vistas y los controladores. Además de mostrar los datos las vistas también suelen incluir enlaces a otras acciones del controlador, para que el cliente pueda invocar otras acciones propias de la aplicación web.

Controladores: son los encargados de ofertar el catálogo de acciones que la aplicación web es capaz de realizar (y que se corresponderá con la lógica de negocio implementada), cuando el usuario selecciona una de estas acciones el controlador debe

ejecutar la lógica de negocio asociada y generar una respuesta, en muchos casos estas respuestas generarán un cambio en la vista actual de la aplicación. Dentro de la lógica de negocio de las acciones del controlador es posible que se acceda a los datos de la aplicación, estos datos no se manipulan directamente, sino que la acción se realiza a través del Modelo.

Un ejemplo de esquema de funcionamiento en una aplicación con arquitectura MVC podría ser el siguiente.



El usuario envía una petición inicial al sitio Web "/inicio", el controlador responde a esa petición "/inicio" ejecutando la acción asociada, dentro de la implementación de la acción del controlador se realiza una petición a la función "getUltimasEntradas" del **Modelo** para obtener las últimas 5 entradas del Blog. El Modelo accede a la base de datos para recuperar la información y se la remite al controlador como respuesta. Cuando el controlador obtiene la respuesta la guarda en una estructura de datos y se la adjunta como parámetro a la **Vista** "MostrarEntradas.html" (con la información de las entradas se completa el HTML), además indica que esta misma vista se debe mostrar al cliente como respuesta la acción que ha realizado. La **Vista** "MostrarEntradas.html" que ha sido seleccionada como respuesta, contiene principalmente el código HTML y CSS que debe mostrar para cada una de las entradas, con un pequeño script recorre la lista de Entradas que le ha sido pasada como parámetro y genera una página HTML / CSS donde se muestra la información de las 5 entradas.

El framework Spring al igual que muchos otros nos provee ya de una arquitectura implementada para utilizar este patrón.

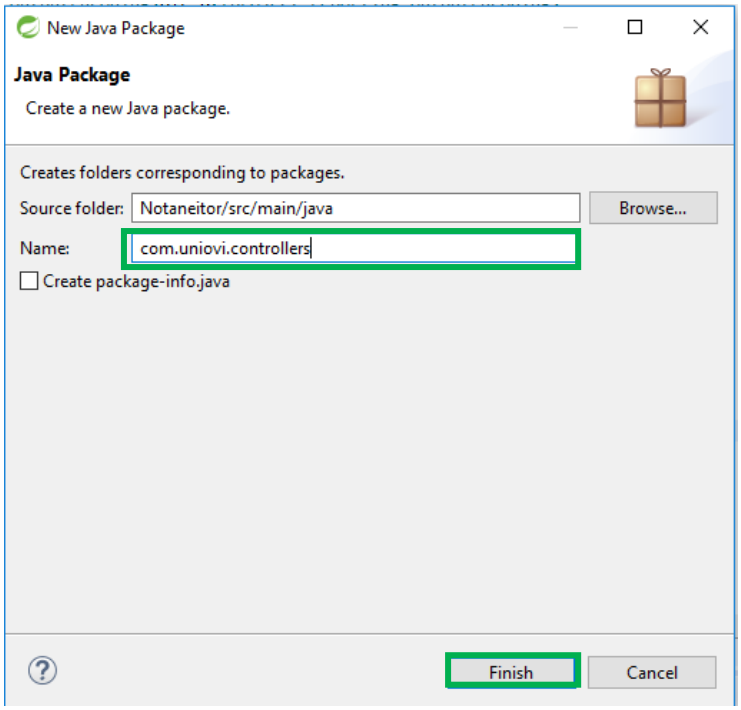
1.8 Controlador

El controlador se encarga de recibir las peticiones del cliente, ejecutar la lógica de negocio correspondiente y generar una respuesta.

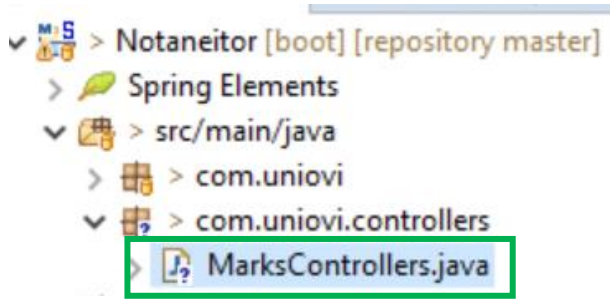
Una aplicación web suele tener al menos un controlador.

Creamos un nuevo paquete java **com.uniovi.controllers** Dentro del paquete creamos la clase Java: **MarksController.java**

Hacemos Click-derecho sobre **el proyecto –> *New -> Package***, nos aparecerá la siguiente ventana. Escribimos el nombre del paquete y hacemos click en Finalizar.



Para crear la clase se sigue el mismo procedimiento anterior, pero colocándonos encima del controlador. Click-derecho sobre ***com.uniovi.controllers –> New -> Class*** y dejamos todos los valores por defecto.



Debemos incluir la anotación *@RestController* en la clase, esto indica que la clase es un controlador (Rest) responde a peticiones Rest (No nos preocupemos por el concepto Rest, ya que lo veremos más adelante). Inicialmente nos dará un error, de que no se puede resolver el tipo *RestController*, porque debemos importar el paquete *org.springframework.web.bind.annotation.RestController*. Para esto, solo hay que modificar la clase como se muestra a continuación.

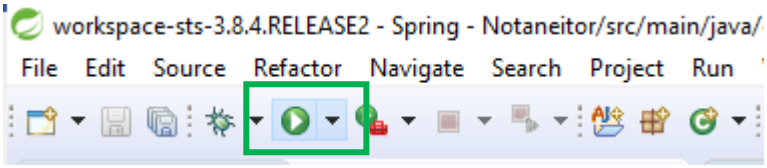
```
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MarksController {

}
```

Ahora, incluimos una función por cada URL a la que va a responder el controlador, la función puede tener cualquier nombre, lo importante es añadir la anotación **@RequestMapping** especificando a que URL responde. Al igual que antes nos dará un error la etiqueta @RequestMapping, por lo que debemos importar el paquete correspondiente.

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MarksController {

	@RequestMapping("/mark/list")
	public String getList(){
		return "Getting List";
	}

	@RequestMapping("/mark/add")
	public String setMark(){
		return "Adding Mark";
	}

	@RequestMapping("/mark/details ")
	public String getDetail(){
		return "Getting Details";
	}

}
```

Ya tenemos una aplicación capaz de responder a 3 URLs, aunque por el momento no ejecuta nada de lógica de negocio, responde simplemente con un texto plano (String).

Ejecutamos la aplicación y probamos a acceder a las 3 URL.



http://localhost:8090/mark/list

http://localhost:8090/mark/add

http://localhost:8090/mark/details

Si la aplicación trata con varias entidades (Notas, asignaturas, alumnos, etc.) lo más correcto es crear diferentes controladores, por ejemplo, uno para notas, otro para asignaturas, otro para alumnos, etc.

Como sabemos las URL pueden ser llamadas utilizando diferentes métodos HTTP, principalmente las aplicaciones web utilizan los métodos GET y POST (aunque también puede haber otros).

Las peticiones GET pueden incluir parámetros solamente en la URL, las POST también pueden incluirlos en el cuerpo de la petición http (body).

Vamos a comenzar especificando que ***/mark/details*** sea una petición GET y que pueda recibir un parámetro id.

Definimos el parámetro que va a recibir utilizando la anotación @**RequestParam** asociada al parámetro **Long id,** esto indica que la URL puede recibir un parámetro con clave "id". Al igual que antes nos dará un error de compilación la etiqueta @RequestParam, por lo que debemos importar el paquete correspondiente.

```
@RequestMapping("/mark/details" )
public String getDetail(@RequestParam Long id){
        return "Getting Detail: "+id;

}
```

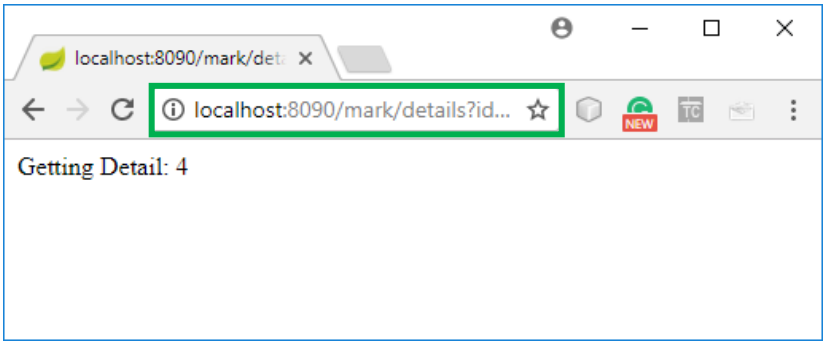
Tal y como lo hemos declarado la petición (Request), puede contener un parámetro con clave "id" (El controlador busca un parámetro en cualquier parte de la URL con clave id).

Detenemos la aplicación y la ejecutamos de nuevo para ver el cambio.

Al realizar peticiones a detalles enviando un parámetro en la URL con clave "id" debería mostrarnos ese mismo parámetro en la respuesta.

http://localhost:8090/mark/details?id=4

http://localhost:8090/mark/details?anotherParam=45&id=4



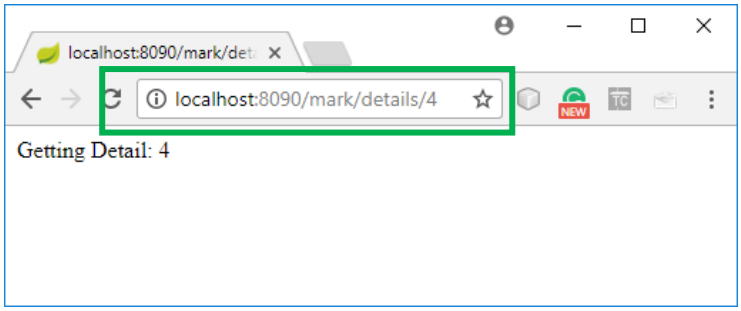
Otra forma común de incluir parámetros GET es hacerlo en el propio Path, sin tener que indicar la clave del parámetro, sino que la clave viene dada por su posición en el Path.

Para ello debemos incluir la posición de la variable en la URL y utilizar la anotación **@PathVariable** en lugar de la anterior.

```
@RequestMapping("/mark/details/{id}" )
public String getDetail(@PathVariable Long id){
        return " Getting Detail: "+id;

}
```

Probamos que este enfoque también funciona de forma correcta.

http://localhost:8090/mark/details/4/



A continuación, vamos a ver cómo tratar peticiones POST, en primer lugar, debemos indicar que la URL admite peticiones POST. Es decir, por defecto estaba activado el GET, si queremos que reciba peticiones POST debemos especificarlo

Como ahora la anotación **@RequestMapping** tiene más de una propiedad debemos especificar cuál es cada una. **value**, es la URL, method es el método http.

```
@RequestMapping(value="/mark/add", method=RequestMethod.POST )
public String setMark(){
    return "Adding Mark";
}
```

Al igual que antes nos dará un error la anotación @ **RequestMapping**, por lo que debemos importar el paquete correspondiente. Si queremos evitar seguir importado las etiquetas se puede importar directamente todo el paquete annotation.

```
import org.springframework.web.bind.annotation.*;
```

Después declaramos los parámetros que pueden venir contenidos en el cuerpo (Body) de la petición POST.

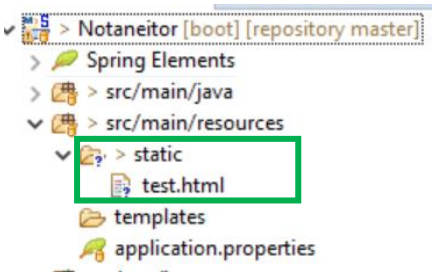
Si los parámetros van a ser enviados a través de un formulario cada uno va a tener una clave y un valor. No obstante, el cuerpo de un body puede estar codificado diversos formatos: "json", "application/x-www-form-urlencoded", "text/plain", para nosotros esto es poco relevante, ya que el framework es capaz de extraer el dato en la mayor parte de formatos.

Sabemos que el body va a contener un parámetro **email** y otro score**(puntuación)**, los agregamos a los argumentos y a la respuesta.

```
@RequestMapping(value="/mark/add", method=RequestMethod.POST )
public String setMark(@RequestParam String description,
                      @RequestParam String score){

        return "Added: "+description+" with score: "+score;
}
```

Ahora tenemos que probar a realizar una petición POST, para hacer una prueba rápida vamos a crear un fichero **test.html** en la carpeta **src/main/resources -> static** del proyecto (esta carpeta está reservada para recursos estáticos: html, imágenes, css, etc).

La carpeta **static** s utiliza para guardar recursos estáticos (normalmente fotografías, css, js) a estos recursos se puede acceder sin necesidad de pasar por el controlador.

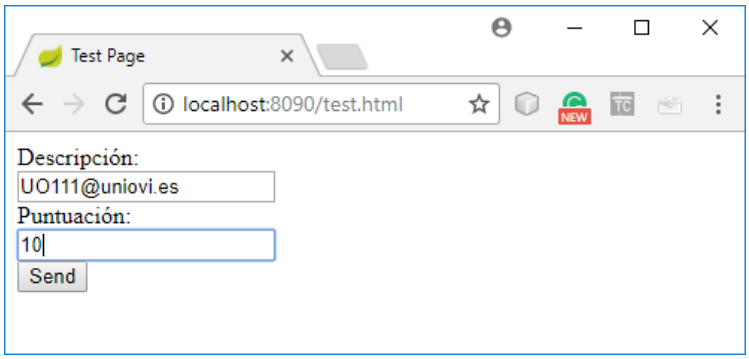


Implementamos un formulario simple que envié mediante **POST** a la URL **/add** los parámetros con las claves **email** y **score**.

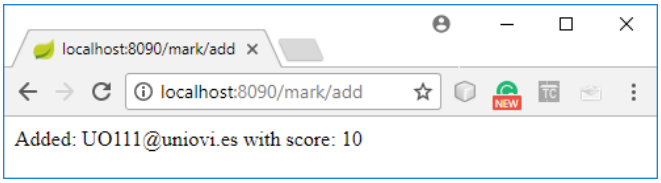
```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>Test Page</title>
        </head>
        <body>
                <form method="post" action="/mark/add">
                  Descripción:<br>
                  <input type="text" name="description"></br>
                  Puntuación:<br>
                  <input type="text" name="score" ></br>
                  <input type="submit" value="Send">
                </form>
        </body>

</html>
```

Detenemos la aplicación y la volvemos a ejecutar, acedemos al recurso estático http://localhost:8090/test.html

Introducimos unos datos de ejemplo.



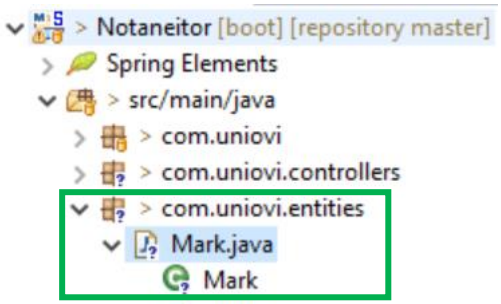
Al pulsar enviar deberíamos mostrarnos la respuesta del controlador a **POST /mark/add**.



Pero existe otra forma más directa de recibir parámetros, cuando todos los parámetros corresponden a una misma entidad podemos mapearlos directamente a un objeto.

Creamos el paquete **com.uniovi.entities** y dentro la clase **Mark.java**

En esta clase incluimos los atributos: **id, email y score**, para cada uno de ellos debemos incluir un método get y set (usar el asistente del STS).



Para usar el asistente hacemos click derecho sobre la clase java y luego vamos a la opción *source->Generate Getters and Setters* y luego seleccionar los atributos para los cuales queremos generar estos métodos. La implementación será la siguiente:

```
package com.uniovi.entities;

public class Mark {
        private Long id;
        private String description;
        private Double score;

        public Long getId() {
```

```
                return id;
        }
        public void setId(Long id) {
                this.id = id;
        }
        public String getDescription() {
                return description;
        }
        public void setDescription(String description) {
                this.description = description;
        }
        public Double getScore() {
                return score;
        }
        public void setScore(Double score) {
                this.score = score;
        }

}
```

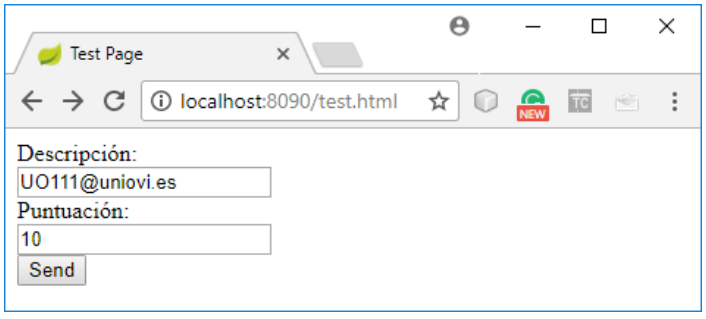
Vamos a modificar los parámetros del controlador **POST /mark/add**. Utilizando la anotación **@ModelAttribute**, podemos mapear los parámetros directamente a un objeto (siempre que los nombres de los parámetros recibidos coincidan con los nombres de los atributos del objeto. Para utilizar el objeto Nota hay que importar el paquete *com.uniovi.entities* dentro del controlador.

```
        @RequestMapping(value="/mark/add", method=RequestMethod.POST )
        public String setMark(@ModelAttribute Mark mark){
                return "added: " + mark.getDescription()
                +" with score : "+ mark.getScore()
                +" id: " + mark.getId();

        }
```

Sí el objeto del modelo en el que se mapean los parámetros tiene más atributos de los recibidos estos atributos no se inicializarán, serán null.

En este caso observaremos ese problema con el atributo id.

Desplegamos la aplicación de nuevo y la probamos:



Nota: Subir el codigo a GitHub en este punto. Commit Message -> ***"SDI - 1.8 Trabajando con Controladores."***

1.9 Beans - Servicios, inyección de dependencias

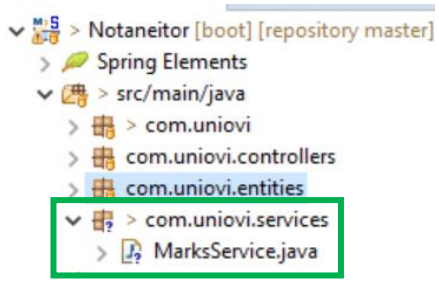
En primer lugar, vamos a ampliar la implementación de la clase Mark(Notas), añadimos un ***constructor con parámetros y un constructor sin parámetros*** (al incluir el "constructor con parámetros" desaparece el constructor implícito sin parámetros y debemos incluirlo explícitamente). Implementamos también el método toString(). En ambos casos podemos utilizar el asistente de STS para generar el código correspondiente. Para usar el asistente hacemos click derecho sobre la clase java y luego vamos a la opción *source->Generate Constructor using Fiel and Generate toString()*. La siguiente imagen muestra el código generado.

```
public Mark(Long id, String description, Double score) {
        super();
        this.id = id;
        this.description = description;
        this.score = score;
}

public Mark(){
}

@Override
public String toString() {
        return "Mark [id=" + id + ", description=" + description + ", score=" + score + "]";

}
```

Vamos a implementar un **Servicio** para gestionar todo lo relativo a la lógica de negocio de las Notas. Los servicios funcionan internamente como Beans, es decir al lanzar el proyecto se crea automáticamente un Bean por cada servicio, estos Beans quedan disponibles para poder ser inyectados y utilizados en otras partes de la aplicación. Particularmente nos va a interesar inyectar y utilizar este servicio en el (**MarksController.java**)

La inyección de dependencias o inversión de control tiene como uno de sus objetivos desacoplar el código entre los diferentes componentes de la aplicación. Por ejemplo, un controlador que responde a peticiones seguramente utilizara funciones de un servicio que contenga la lógica de negocio de la aplicación, para relacionarse con ese servicio no crea una instancia del servicio, sino que únicamente define una dependencia con el servicio y lo utiliza (no se encargan de crearlo, simplemente lo inyecta y lo utiliza).

Creamos el paquete **com.uniovi.services**, en el creamos la clase **MarksService.java**



La anotación **@Service** indica que esta clase es un servicio. Si queremos que una función actué como "inicializador" debemos incluir la anotación **@PostConstruct,** (en este caso lo vamos a utiliza en la función init). Para poder utilizar estas anotaciones tenemos que

importar los paquetes ***import org.springframework.stereotype.Service*** y ***import javax.annotation.PostConstruct,*** Respectivamente.

```
@Service
public class MarksService {

        private List<Mark> marksList =  new LinkedList<Mark>();

    @PostConstruct
    public void init(){
        marksList.add(new Mark(1L,"Ejercicio 1",10.0));
        marksList.add(new Mark(2L,"Ejercicio 2",9.0));
    }

        public List<Mark> getMarks(){
                return marksList;
        }

        public Mark getMark(Long id){
                return marksList.stream()
                                .filter(mark -> mark.getId().equals(id)).findFirst().get();
        }

        public void addMark(Mark mark){
                // Si en Id es null le asignamos el ultimo + 1 de la lista
           if(mark.getId() == null) {
                   mark.setId(marksList.get(marksList.size() - 1).getId() + 1);
           }

           marksList.add(mark);
        }

        public void deleteMark(Long id){
                marksList.removeIf(mark -> mark.getId().equals(id));
        }
}
```

En alguna ocasión en auto-importador del eclipse puede no sugerirnos importar clases de nuestro propio proyecto, si nos ocurre esto incluimos la importación de forma manual, por ejemplo, para importar nota agregamos manualmente:

```
import com.uniovi.entidades.Mark;
```

¿Cómo podemos declarar que el servicio MarksService va a ser usado desde el controlador? Lo vamos a inyectar creando una variable dentro del controlador y utilizando la anotación **@Autowired.**

@Autowired. Se utiliza dentro del Framework Spring para inyectar beans. Esta etiqueta busca Bean correspondiente en el proyecto. (Los **@Service** crean un Bean). Aunque de momento no vamos a utilizarlas también existen otras etiquetas para inyectar beans de forma un poco más específica, como **@Resource o @Inject**. Para usar **@Autowired** hay que importar el paquete ***org.springframework.beans.factory.annotation.*****Autowired**; En esta URL se explica la diferencia entre las distintas anotaciones:

- http://www.notodocodigo.com/spring/conexion-automatica-de-beans
- https://blogs.sourceallies.com/2011/08/spring-injection-with-resource-and-autowired/

```
@RestController
public class MarksController {
```

```
@Autowired //Inyectar el servicio
private MarksService marksService;
```

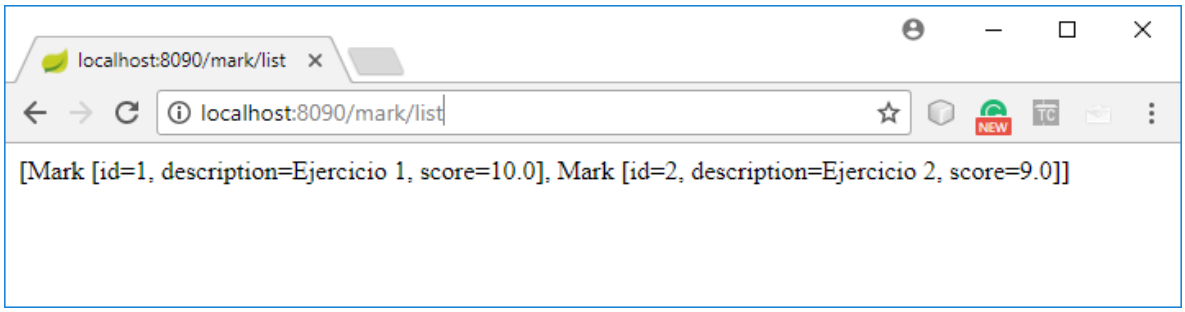
Modificamos la implementación de las funciones del controlador para que utilicen la lógica de negocio implementada en el Servicio. Aprovechamos la modificación para agregar una respuesta a la nueva **URL /mark/delete/{id}**

```
@RequestMapping("/mark/list")
public String getList(){
        return marksService.getMarks().toString();
}

@RequestMapping(value="/mark/add", method=RequestMethod.POST )
public String setMark(@ModelAttribute Mark mark){
        marksService.addMark(mark);
        return "Ok";
}

@RequestMapping("/mark/details/{id}" )
public String getDetail(@PathVariable Long id){
        return marksService.getMark(id).toString();
}

@RequestMapping("/mark/delete/{id}" )
public String deleteMark(@PathVariable Long id){
        marksService.deleteMark(id);
        return "Ok";

}
```

Reiniciamos la aplicación y comprobamos que continúa funcionando de la forma esperada. Podemos ver que se pueden **añadir, eliminar y lista** las notas. http://localhost:8090/mark/list

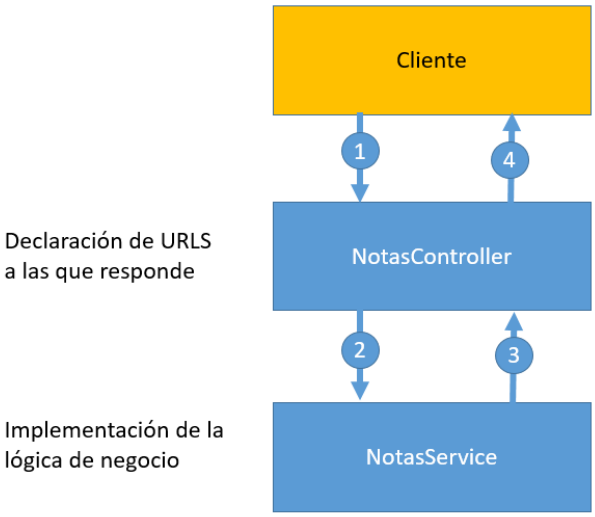
http://localhost:8090/mark/details/1

http://localhost:8090/mark/delete/1



El servicio se instancia una única vez en la creación de la aplicación, puede ser inyectado e utilizado en cualquier parte de la aplicación (Java) incluyéndolo de la misma manera, con la anotación **@Autowired** en cada uno de los componentes que quiera utilizar.

Después de está modificación la arquitectura de la aplicación ha pasado a tener un componente nuevo:



Nota: Subir el codigo a GitHub en este punto. Commit Message -> ***"SDI - 1.9 Beans - Servicios, inyección de dependencias"***

1.10 Modelo - Acceso a datos Simple

En la versión anterior hemos utilizado una lista en memoria como repositorio para almacenar las notas, vamos a modificar la aplicación para utilizar una base de datos externa.

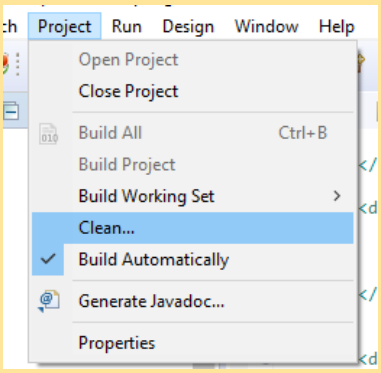
Agregamos al fichero **pom.xml** las dependencias de **JPA** y **HSQB**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>

</dependency>
```

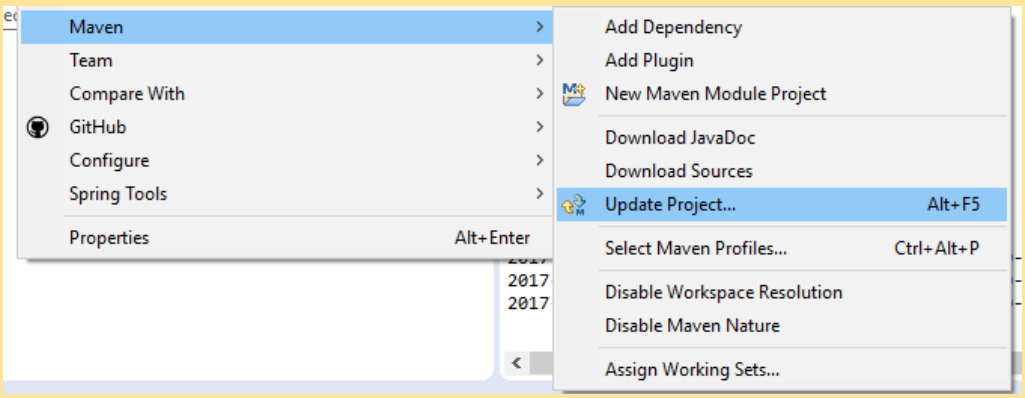
Al guardar los cambios en el fichero se debería actualizar el workspace (e incluir las librerías en el proyecto).

Sí en algún momento el Maven nos da problemas podemos hacer un **Clean** del proyecto y volver a actualizarlo manualmente

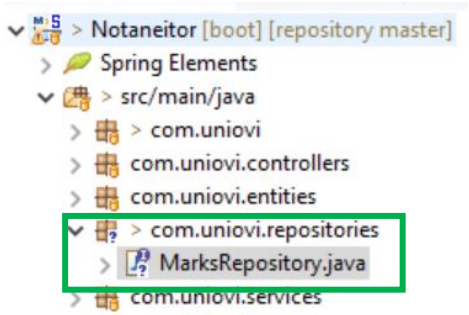
Clean: seleccionar el nombre del proyecto, menú superior **Project -> Clean**.



Actualizar el Maven manualmente, botón derecho sobre el nombre del **proyecto -> Maven -> Update Project.**



Creamos el paquete **com.uniovi.repositories**, y dentro de él la **Interfaz MarksRepository**.



No vamos a implementar manualmente el acceso a datos, sino que nos serviremos de la interfaz **CrudRepository<Clase Entidad, Clase de la clave primaria>** de Spring que incluye toda la implementación para acceso a datos de tipo CRUD.

```
import org.springframework.data.repository.CrudRepository;

public interface MarksRepository extends CrudRepository<Mark, Long>{

}
```

Para utilizar CrudRepository tenemos que importar el paquete ***org.springframework.data.repository.CrudRepository.*** Al haber utilizado el **CrudRepository** esta clase ya es un Bean (es decir lo podemos inyectar en cualquier componente, posteriormente lo inyectaremos en el servicio ***MarksService***, ya que ahí donde queremos utilizarlo).

Nos falta incluir las anotaciones de JPA en la clase com.uniovi.entities.Mark(Notas) para que sea reconocida por JPA como una entidad de repositorio. Indicamos que se mapeara como una entidad **@Entity**, y que su clave primaria es el atributo id (usamos las anotaciones **@Id** y **@GeneratedValue** para generar automáticamente las ids).

Importar el paquete javax.persistence.

```
import javax.persistence.*;

@Entity
public class Mark {
        @Id
        @GeneratedValue
        private Long id;
        private String description;
        private Double score;
```

Volvemos al Servicio MarksService y dejamos de utilizar por completo la lista en memoria para almacenar las notas, comenzaremos a utilizar el **MarksRespository**.

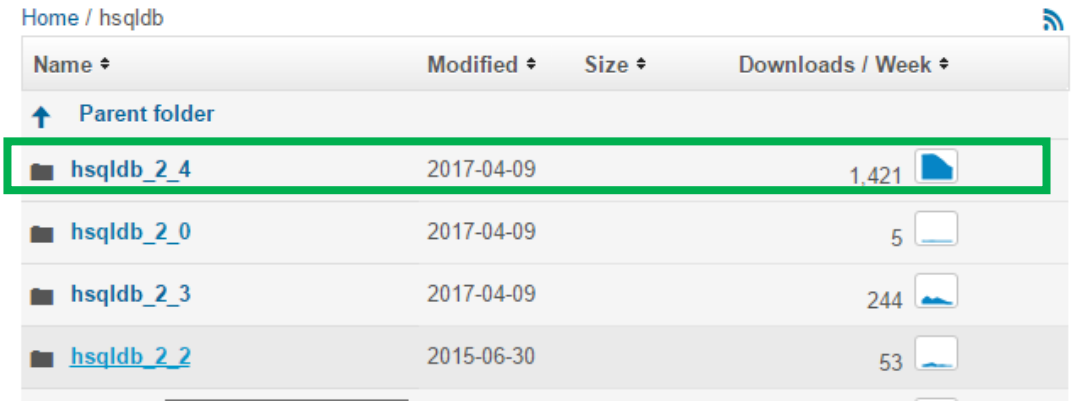
En primer lugar, lo inyectamos y después cambiamos la implementación de las funciones. Eliminamos el uso de la angitua lista marksList.

```
public class MarksService {

        @Autowired
        private MarksRepository marksRepository;

        private List<Mark> marksList = new LinkedList<Mark>();
```

```
    @PostConstruct
    public void init(){
        marksList.add(new Mark(1L,"UO111@uniovi.es",10.0));
        marksList.add(new Mark(2L,"UO222@uniovi.es",9.0));
    }

        public List<Mark> getMarks(){
                List<Mark> marks = new ArrayList<Mark>();
                marksRepository.findAll().forEach(marks::add);
                return marks;
        }

        public Mark getMark(Long id){
                return marksRepository.findById(id).get();
        }

        public void addMark(Mark mark){
                // Si en Id es null le asignamos el ultimo + 1 de la lista
                marksRepository.save(mark);
        }

        public void deleteMark(Long id){
                marksRepository.deleteById(id);

        }
```

Ahora solo nos queda desplegar un servidor de la base de datos HSQLDB en nuestra maquina e incluir las propiedades de conexión en el fichero **application.properties** del proyecto.

Accedemos al sitio web de hsqldb https://sourceforge.net/projects/hsqldb/files/hsqldb/ y descargamos la versión **hsqldb_2_4**



Descomprimimos el fichero descargado y ejecutamos el fichero **hsqldb -> bin -> runServer.**bat. La versión para MACOSX/Linux la puedes crear asi:

$touch runServer.sh

$chmod 755 runServer.sh

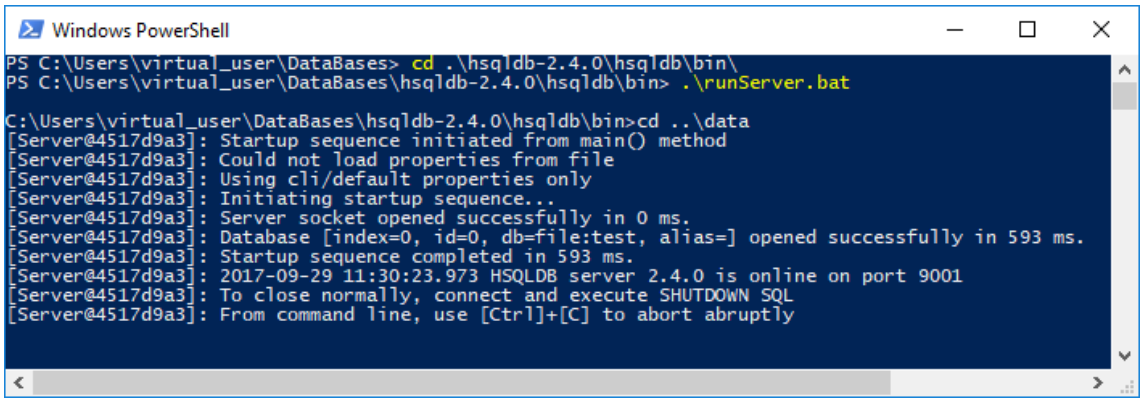
Ahora usando tu editor favorito adjunta el siguiente código Shell:

cd ../data

java -classpath ../lib/hsqldb.jar org.hsqldb.server.Server $1 $2 $3 $4 $5 $6 $7 $8 $9

Y para lanzarla:

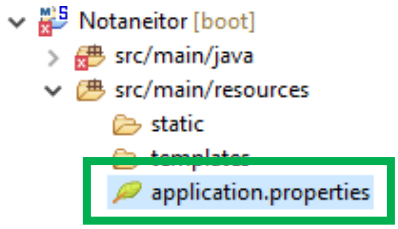
$sh runServer.sh



Debemos recordar lanzar la base de datos siempre que ejecutamos la aplicación por primera vez, ya que este servidor no se arranca automáticamente al desplegar la aplicación.

Si cerramos la ventana de comandos el servidor de la base de datos se cerrará.

Abrimos el fichero **aplication.properties** de la aplicación e introducimos las propiedades de conexión:




```
server.port = 8090

spring.datasource.url=jdbc:hsqldb:hsql://localhost:9001
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver

spring.jpa.hibernate.ddl-auto=create
# No crear ninguna tabla, solo validar:
#spring.jpa.hibernate.ddl-auto=validate
# Crear la tabla de nuevo

#spring.jpa.hibernate.ddl-auto=create
```

La ultima propiedad `spring.jpa.hibernate.ddl-auto=create` indica que debe crear el modelo de datos, es decir crear la tabla Marks (Notas), con los campos indicados. Si mantenemos este atributo cada vez que ejecutamos la aplicación se van a crear las tablas (es decir vamos a perder todos los datos).

Nos aseguramos de guardar todos los cambios, detenemos y volvemos a arrancar la aplicación.

Probamos a ejecutar la aplicación.

1. http://localhost:8090/mark/list consultamos el listado, debería estar vacío
2. http://localhost:8090/test.html agregamos al menos dos notas.
3. http://localhost:8090/mark/list volver a consultar el listado
4. http://localhost:8090/mark/details/1 ver detalles de la nota id=1
5. http://localhost:8090/mark/delete/1 eliminar la nota con id=1
6. http://localhost:8090/mark/list la nota id=1 no debe figurar

Si nos interesa mantener los datos una vez estemos seguros de que la base de datos ha sido creada cambiamos el valor del atributo a `spring.jpa.hibernate.ddl-auto=validate`

Nota: Subir el codigo a GitHub en este punto. Commit Message -> ***"SDI - 1.10 Modelo - Acceso a datos Simple"***

1.11 Vistas – Motor de plantillas thymeleaf

Existen varios motores de plantillas que se usan de forma más habitual en Spring (y otros frameworks ya que son independientes), uno de los más populares es http://www.thymeleaf.org .

¿Por qué usar un motor de plantillas?

- Sintaxis sencilla
- No solo insertan datos y contiene estructuras de control básicas (if, for, expresiones lógicas, etc.), Sino que algunos posen componentes muy avanzados para incluir en las vistas: sistemas de validación, fragmentos de ajax, internacionalización, división de plantillas en bloques, etc.
- Reutilizable, separación total entre la vista y la lógica, además el mismo motor de plantillas puede ser utilizado en diferentes frameworks.

El framework Spring puede combinarse con varios motores de plantillas diferentes: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/view.html en la actualidad uno de los más populares evolucionados y potentes es thymeleaf

En primer lugar debemos incluir la nueva dependencia a thymeleaf, en el fichero **pom.xml**

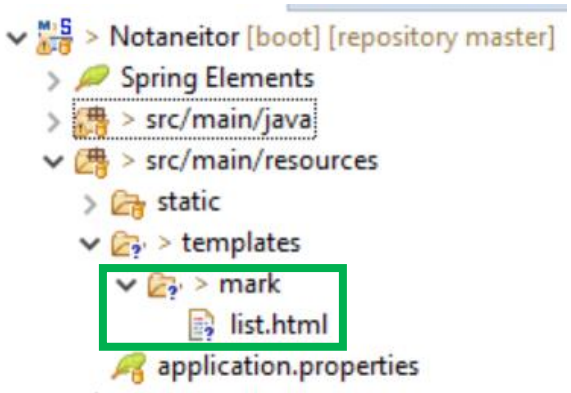
```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Debemos recordar salvar el fichero después de agregar la dependencia.

La mayor parte de los motores de plantillas almacenan sus plantillas por defecto en el directorio **src/main/resources/templates**

Podemos crear una carpeta llamada **/mark/** para tener una organización mejor.

Dentro creamos un fichero **list.html**



El contenido de la template **list.html** será el siguiente:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Notaneitor</title>
<meta charset="utf-8" />
</head>
<body>

<div class="container">
    <h1>List Thymeleaf</h1>
    <div th:each="mark : ${markList}">
        <p>
            <span th:text="${mark.id}"></span>
            <span th:text="${mark.description}"></span>
            <span th:text="${mark.score}"></span>
            <a th:href="${'/mark/details/' + mark.id}">details</a>
            <a th:href="${'/mark/delete/' + mark.id}">delete</a>
        </p>
    </div>

    <div th:if="${#lists.isEmpty(markList)}">
        No marks
    </div>
</div>

</body>
</html>
```

La plantilla espera recibir una lista **markList** con objetos de tipo **mark**, cada uno con **id**, **description** y **score**.

En este caso estamos utilizando 4 de las etiquetas más comunes del motor, estas etiquetas se deben asociar a un control de HTML

- th:each="mark : ${markList}": itera la lista de notas guardando en el objeto ***mark*** la nota recorrida en cada iteración. Como está asociado al componente div de HTML en cada iteración va a crear un con su correspondiente contenido.
- th:text="${mark.id}: inserta un nuevo texto en el componente HTML al que se asocial, significa que dentro del va a incluir el texto #Texto mark.id Puede ser usado con cualquier componente HTML para incluir texto dentro de él, un div, un párrafo, una lista, etc.
- th:href="${'/details/' + mark.id}": inserta un nuevo atributo href en el componente HTML al que se asocia, en este caso está asociado a un enlace <a> por lo que va a incluir un nuevo atributo href=”/details/1” (suponiendo que el mark.id fuera 1.
- th:if="${#lists.isEmpty(markList)}: evalúa una expresión lógica si la expresión se cumple incluye el div el la página. En este caso se está valiendo de la función de ayuda **#list.isEmpty** para comprobar si la lista está vacía.

La raíz de la carpeta de las plantillas es **/templates/**, todos los directorios que creemos dentro de esa carpeta debemos especificarlos en el nombre de la vista cuando queramos retornarlas en el controlador.

Para que el controlador pueda responder con vistas debemos cambiar la anotación inicial **@RestController**, por **@Controller, ya que la anotación @Controller especifica una respuesta con contenido HTML y no REST como en el caso de @RestController.**

```
@Controller
public class MarksController {
```

¿Cómo podemos "completar" y "retornar" las vistas desde el controlador? El controlador necesita crear un modelo de datos (clase Model) e insertar dentro de ese modelo el atributo **markList** que espera recibir la plantilla **mark/list.html**

Después de insertar el atributo, el controlador debe retornar como String la clave de la vista que quiere mostrar (las claves son las rutas de los ficheros desde la carpeta /templates, en este caso **mark/list**).

Realizamos los cambios en el argumento de la función **Model model,** y el cuerpo. Hay que importar el paquete ***org.springframework.ui.Model***;

```
@Controller
public class MarksController {

	@Autowired //Inyectar el servicio
	private MarksService marksService;

	@RequestMapping("/mark/list")
	public String getList(Model model){
		model.addAttribute("markList", marksService.getMarks() );
		return "mark/list";
	}
```

Otra cosa que debemos tener en cuenta que, al hacer el cambio a motor de vistas, los métodos que utilizamos anteriormente en el controlador no van a funcionar correctamente, puesto que no retornan ninguna vista.

Por ejemplo:

```
@RequestMapping(value="/add", method=RequestMethod.POST )
public String setNota(@ModelAttribute Nota nota){
	notasService.addNota(nota);
	return "Ok";

}
```

La respuesta de esta función hace que la aplicación intente responder un fichero con el contenido: "**Ok" (Es sensible a mayúsculas).**

Finalmente incluimos una redirección a la URL list desde las funciones **setMark** y **deleteMark.**

```
@RequestMapping(value="/mark/add", method=RequestMethod.POST )
public String setMark(@ModelAttribute Mark mark){
        marksService.addMark(mark);
        return "redirect:/mark/list";
}

@RequestMapping("/mark/details/{id}" )
public String getDetail(@PathVariable Long id){
        return marksService.getMark(id).toString();
}

@RequestMapping("/mark/delete/{id}" )
public String deleteMark(@PathVariable Long id){
        marksService.deleteMark(id);
        return "redirect:/mark/list";

}
```

Guardamos los cambios y detenemos / arrancamos la aplicación de nuevo.

Problema: Por defecto debemos tener cuidado cuando estamos implementado las plantillas ya que los fallos de sintaxis en el HTML pueden hacer que la plantilla no llegue a compilar.

Por ejemplo, un sin cerrar, puede provocar un error de parseo. (Probamos a quitar uno de los de la plantilla list.html).

Error en consola:

```
org.xml.sax.SAXParseException: El tipo de elemento "div" debe finalizar por la etiqueta final coincidente "</div>".
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException(Unknown Source) [na:1.8.0_91]
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalError(Unknown Source) [na:1.8.0_91]
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError(Unknown Source) [na:1.8.0_91]
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError(Unknown Source) [na:1.8.0_91]
        at com.sun.org.apache.xerces.internal.impl.XMLScanner.reportFatalError(Unknown Source) [na:1.8.0_91]
```

También algunas etiquetas que en HTML tradicional admiten ser utilizadas sin cierre pueden causar errores en el parseo de la plantilla, como la etiqueta **Meta** o **Link** que aunque normalmente se utilicen sin cierre aquí debe incluirse. Por lo tanto usaremos:

```
<meta charset="utf-8" />
<link rel="stylesheet" href=" bootstrap.min.css"/>
```

En lugar de

```
<meta charset="utf-8" >
<link rel="stylesheet" href=" bootstrap.min.css">
```

Nota: Apartir de la versión 2.X.X de Spring Boot este problema se ha solucionado.

No obstante si el problema ocurre, existe una librería que podemos agregar a nuestro proyecto para que thymeleaft deje de ser "estricto", incluimos la dependencia en el **pom.xml**

```
<dependency>
     <groupId>net.sourceforge.nekohtml</groupId>
     <artifactId>nekohtml</artifactId>
     <version>1.9.21</version>
 </dependency>
```

Una vez agregada debemos incluir la siguiente línea en el fichero **application.properties.**

```
spring.thymeleaf.mode=LEGACYHTML5
```

Guardamos todos los cambios y la aplicación dejará de ser estricta al procesar las plantillas

Vamos a sustituir la plantilla **list.html** por otra versión que utiliza el framework UI bootstrap (Copiar el contenido del fichero **list.html descargado del campus virtual o de la URL de los recursos**).

Si examinamos el fichero vemos que está utilizando las mismas etiquetas que en la plantilla anterior, pero en este caso incluye una fila de la tabla **<tr>** por cada nota, con cinco columnas **<td>**.

```
<tbody>
  <tr th:each="mark : ${markList}">
    <td th:text="${mark.id}"> 1</td>
    <td th:text="${mark.email}"> email@email.com</td>
    <td th:text="${mark.score}">10</td>
    <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
    <td><a th:href="${'/mark/edit/' + mark.id}">modificar</a></td>
    <td><a th:href="${'/mark/delete/' + mark.id}">eliminar</a></td>
  </tr>
</tbody>
```

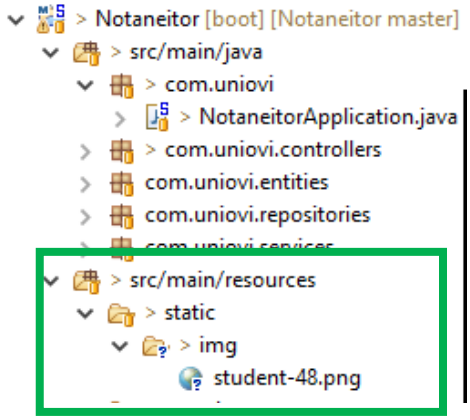
Si ejecutamos la aplicación de nuevo veremos la nueva plantilla. La imagen de la parte superior izquierda no se ha cargado correctamente.

Según el código HTML la imagen debería estar en **img/student-48.png**

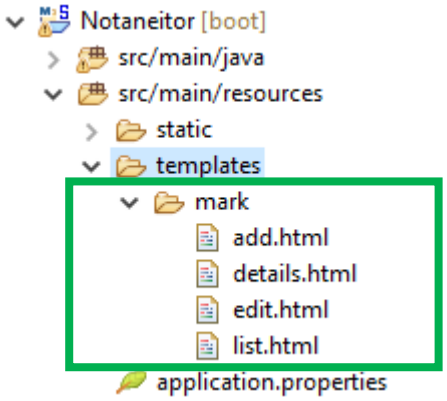
```
<img src="/img/student-48.png" alt="Logo"/>
```

Debemos recordar que es mejor nombrar los recursos con paths absolutos.

Todos los recursos van a empezar a buscarse en la parte **resources/static/** de nuestro proyecto, por lo tanto, es ahí donde debemos crear la carpeta **/img/** y copiar la fotografía.



Copiamos las siguientes plantillas en el proyecto: **add, details, edit.**



En el controlador creamos una respuesta para **GET /mark/add**, que responderá directamente con la vista **mark/add**.

```
@RequestMapping(value="/mark/add")
public String getMark(){
        return "mark/add";

}
```

Modificamos la respuesta de **GET /mark/details/{id}.** El método debe recibir el modelo, solicitamos a marksService la nota con el id recibido y guardamos esa nota en el **modelo** bajo la clave "nota", después retornamos la vista ***mark/details.***

```
@RequestMapping("/mark/details/{id}")
public String getDetail(Model model, @PathVariable Long id){
```

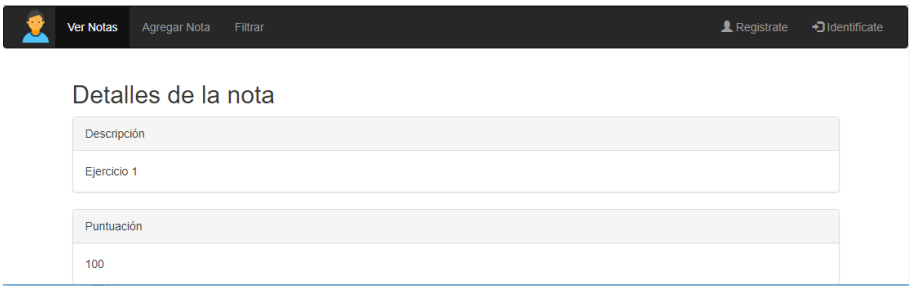
```
            model.addAttribute("mark", marksService.getMark(id));
            return "mark/details";

        }
```

Por lo tanto, la vista **details.html** puede hacer uso del objeto **${mark},** aunque esta parte no se ha incluido en la plantilla, debemos abrir la plantilla **details.html** y modificarla

```
<div class="container">
  <h2>Detalles de la nota</h2>
  <div class="panel panel-default">
      <div class="panel-heading">Descripción</div>
      <div class="panel-body" th:text="${mark.description}" >
        UO11111@uniovi.es
      </div>
  </div>
  <div class="panel panel-default">
      <div class="panel-heading">Puntuación</div>
      <div class="panel-body" th:text="${mark.score}">
        100
      </div>
  </div>
</div>
```

Ejecutamos la aplicación y comprobamos que tanto el **/add** como los **/details** funcionan de forma correcta, el **/delete** también debería funcionar ya que no requería ninguna modificación.

http://localhost:8090/mark/list



Nos falta incluir la capacidad de responder a la petición para editar **/edit** en nuestro controlador, realmente debemos responder dos veces a esta petición:

- **GET /mark/edit/{id}**: retornar una vista sobre la que el usuario va a poder realizar la modificación de los valores actuales
- **POST /mark/edit/{id}:** salvar los nuevos datos , después podemos volver a dirigir al usuario a los detalles de la misma nota, para que pueda ver las modificaciones.

Agregamos respuesta para las dos nuevas URLs en **MarksController**.

La más sencilla será **GET /mark/edit/{id},** buscamos la nota que encaja con la id que recibimos como parámetro, guardamos esa nota como atributo del modelo, retornamos la plantilla **marks/edit** .

```
    @RequestMapping(value="/mark/edit/{id}")
    public String getEdit(Model model, @PathVariable Long id){
```

```
                model.addAttribute("mark", marksService.getMark(id));
                return "mark/edit";

        }
```

Ahora tenemos que hacer uso del objeto nota "mark" en la plantilla **edit.html**. Dentro del contenedor ya tenemos un formulario preparado, pero debemos hacer los siguientes cambios:

- **Form action.** debe enviarse a **/mark/edit/{id}**, para modificar el atributo action de un formulario en thymeleaf se utiliza el atributo **th:action**
- **Input value.** Los valores asociados al input "email" y "puntuación", queremos que se comience marcando los valores que la nota tiene actualmente, utilizamos el atributo de thymeleaf **th:value** para modificar los value de los dos inputs.

Modificamos la plantilla **edit.html**

```
<div class="container">
  <h2>Editar Nota</h2>
  <form class="form-horizontal" method="post" th:action="${'/mark/edit/' + mark.id}" >

    <div class="form-group">
      <label class="control-label col-sm-2" for="email">Email:</label>
      <div class="col-sm-10">
        <input th:value="${mark.description}" type="text"
             class="form-control"
             name="description" placeholder="Ejemplo Ejercicio 1"
             required="true" />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="pwd">Puntuación:</label>
      <div class="col-sm-10">
        <input th:value="${mark.score}" type="number"
            class="form-control"
            name="score" placeholder="Entre 0 y 10"
            required="true" />
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-10">
        <button type="submit" class="btn btn-primary">Modificar</button>
      </div>
    </div>
  </form>
</div>
```

La última modificación que debemos agregar al controlador es responder a POST **/mark/edit/{id}**, es justo donde se van a enviar los datos cuando el usuario pulse el botón de envió del formulario.

Recibimos como parámetro en la URL la id de la nota que está siendo modificada, como **ModelAttribute,** recibimos los atributos de nota que figuren en el formulario (email y puntuación). La misma función addMark del servicio **MarksService** nos sirve para actualizar, (crear -> si el recurso tiene un id nueva, actualiza -> si le enviamos un id que ya está en la tabla, este funcionamiento es el que viene implementado **CrudRepository**, en la cual nosotros habíamos "delegado" la implementación de **NotasRepository** usando su función **save()**).

Le asignamos el id que llega en el Path a la Nota que llega en el post (como resultado del formulario) y después agregamos esa nota. Como el id coincidirá con una nota existente se actualizará en lugar de crearse de nuevo (tampoco retorna error de Id duplicada, simplemente actualiza).

```
@RequestMapping(value="/mark/edit/{id}", method=RequestMethod.POST)
public String setEdit(Model model, @PathVariable Long id, @ModelAttribute Mark mark){
        mark.setId(id);
        marksService.addMark(mark);
        return "redirect:/mark/details/"+id;
}
```

Ejecutamos la aplicación y comprobamos que el modificar funciona de forma correcta.

Otra posible alternativa para realizar el modificar, es incluir un campo oculto en el formulario que tengan la id de la nota (por supuesto no modificable). Aunque no es una opción recomendable.

De esta forma el **"`@ModelAttribute Mark mark`"** que se recibe como parámetro ya va a tener la id asignada, no sería necesario recibirla como parámetro en la URL.

Nota: Subir el codigo a GitHub en este punto. Commit Message -> ***"SDI - 1.11 Vistas – Motor de plantillas thymeleaf."***

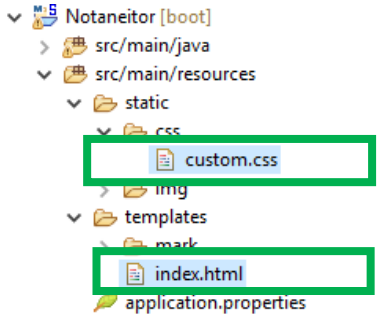
1.12 Página de inicio

Actualmente si accedemos a la raíz de nuestra página web http://localhost:8090 nos dará un error porque no tenemos configurada la página de inicio

Vamos a definir un nuevo controlador **HomeController** para gestionar la página de inicio

```
@Controller
public  class HomeController {

        @RequestMapping("/" )
        public String index() {
                return "index";
        }
}
```

Creamos la plantilla **/templates/index.html** y el estilo, static**/css/custom.css**



El contenido de **index.html** será en primer lugar muy simple, luego lo completaremos mediante el uso de fragmentos. Probamos de nuevo la nuestra web desde http://localhost:8090/

```
<!DOCTYPE html>
<html lang="en">
<body>
<div class="container" style="text-align: center">
        <h2>Bienvenidos a la página principal</h2>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing</p>
</div>
</body>
</html>
```

El contenido de **custom.css** será el siguiente:

```
/* Sticky footer styles-------------------------------------------------- */
html {
  position: relative;
  min-height: 100%;
}
body {
  /* Margin bottom by footer height */
  margin-bottom: 60px;
}
footer {
  position: absolute;
  bottom: 0;
  width: 100%;
  height: 60px;
```

```
  background-color: #101010;
  text-align:center;
  line-height:60px
}
/* Sticky footer styles-------------------------------------------------- */
```

1.13 Diseño de plantillas con Thymeleaf

Por lo general, las interfaces de los sitios comparten componentes comunes: el encabezado, pie de página, menú y posiblemente muchos más. En Thymeleaf hay dos estilos principales de organización de diseños en los proyectos: ***include style y hierarchical style.***

Include-style layouts

En este estilo, las páginas se construyen insertando código de componente de página común directamente en cada vista para generar el resultado final. En Thymeleaf esto se puede hacer usando: ***Thymeleaf Standard Layout System.***

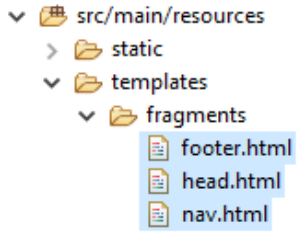
Los diseños de inclusión de estilo son bastante sencillos de entender e implementar y de hecho ofrecen flexibilidad en el desarrollo de puntos de vista, que es su mayor ventaja. La principal desventaja de esta solución, sin embargo, es que se introduce una cierta duplicación de código, por lo que modificar el diseño de un gran número de vistas en grandes aplicaciones puede volverse incómodo.

Hierarchical-style layouts (estilo jerárquico***)***

En el estilo jerárquico, las plantillas suelen crearse con una relación padre-hijo, desde la parte más general (diseño) hasta las más específicas (subviews, por ejemplo, el contenido de la página). Cada componente de la plantilla puede incluirse dinámicamente basándose en la inclusión y sustitución de fragmentos de plantilla. En Thymeleaf esto se puede hacer usando el ***Thymeleaf Layout Dialect***(Dialecto de disposición ***Thymeleaf***).

Las principales ventajas de esta solución son la reutilización de porciones atómicas de la vista y el diseño modular, mientras que la principal desventaja es que se necesita mucha más configuración para poder usarlas, por lo que la complejidad de las vistas es mayor que con Include Style Layouts que son más 'naturales' de usar.

A continuación vamos a crear la carpeta **templates/fragments** y dentro de ella los fragmentos de códio "común", **head.html**, **nav.html** y **footer.html.**




```
<!-- head.html -->
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <script
        src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js">
    </script>
    <script
        src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
```

```
    </script>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" />
    <link rel="stylesheet" href="/css/custom.css" />

</head>
```

```
<!-- nav.html -->
<nav class="navbar navbar-inverse">
     <div class="container-fluid">
         <div class="navbar-header">
             <button type="button" class="navbar-toggle"
                 data-toggle="collapse" data-target="#myNavbar">
                 <span class="icon-bar"></span>
                 <span class="icon-bar"></span>
                 <span class="icon-bar"></span>
             </button>
             <img src="/img/student-48.png" alt="Logo" />
         </div>
         <div class="collapse navbar-collapse" id="myNavbar">
             <ul class="nav navbar-nav">
                 <li class="active"><a href="/mark/list">Ver Notas</a></li>
                 <li><a href="/mark/add">Agregar Nota</a></li>
                 <li><a href="/mark/filter">Filtrar</a></li>
             </ul>
             <ul class="nav navbar-nav navbar-right">
                 <li>
                     <a href="/signup">
                         <span class="glyphicon glyphicon-user"></span>
                         Registrate
                     </a>
                 </li>
                 <li>
                     <a href="/login">
                         <span class="glyphicon glyphicon-log-in"></span>
                         Identifícate
                     </a>
                 </li>
             </ul>
         </div>
     </div>
 </nav>
```

```
<!-- footer.html -->
<footer class="footer">
    <div class="container">
        <span class="text-muted">&copy; SDI - Gestión de notas</span>
    </div>
</footer>
```

Ahora modificamos la plantilla **index.html** para incluir los tres fragmentos "fragmentos" header y footer. Thymeleaf puede incluir partes de otras páginas como fragmentos usando ***th:insert*** (simplemente insertará el fragmento especificado como el cuerpo de su etiqueta de host) o ***th:replace*** (sustituirá el contenido de la página principal en caso de que lo hubiera utilizando el contenido especificado en el fragmento).

```
<!DOCTYPE html>
<html lang="en">
<head th:replace="fragments/head"/>
<body>

<nav th:replace="fragments/nav"/>

<div class="container" style="text-align: center">
        <h2>Bienvenidos a la página principal</h2>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing</p>
</div>

<footer th:replace="fragments/footer"/>

</body>
</html>
```

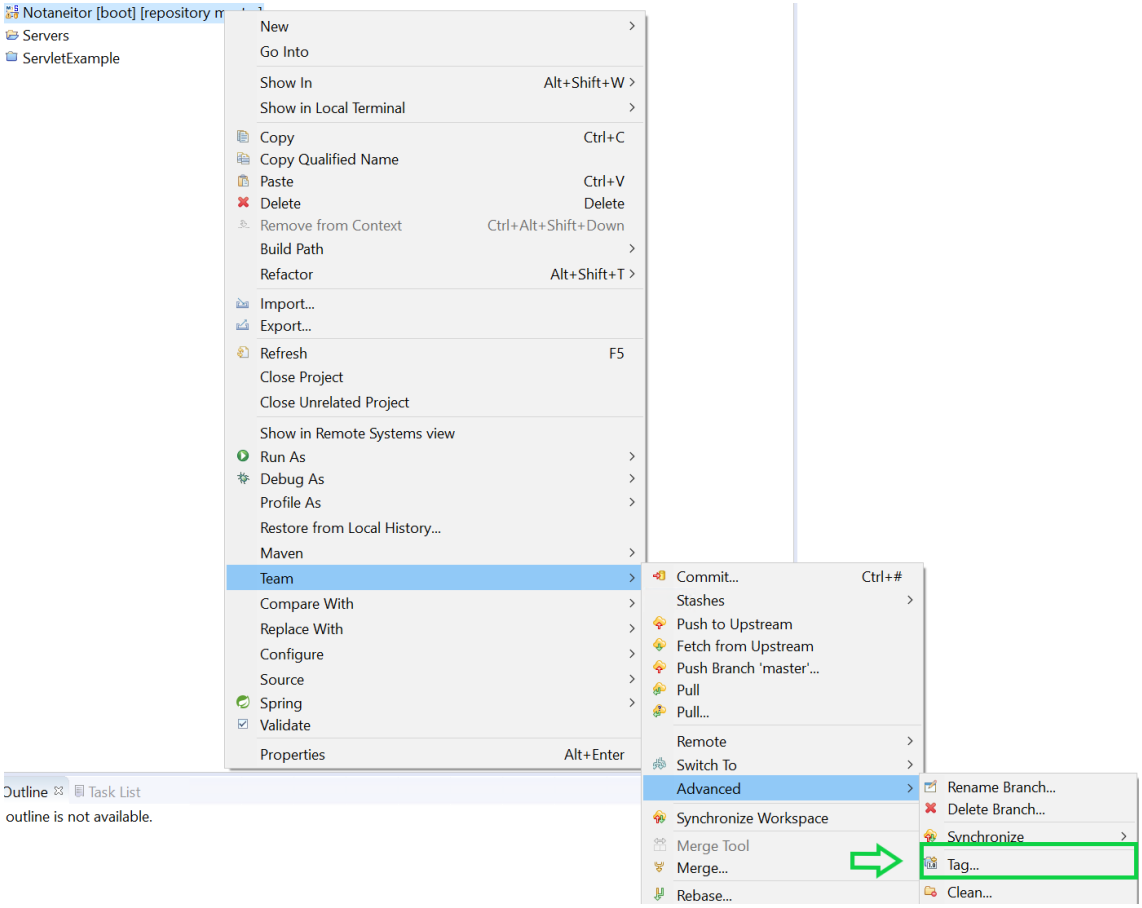
Resultado final de la página principal de la aplicación Web.

Nota: Subir el codigo a GitHub en este punto. Commit Message -> ***"SDI - 1.12 Página de inicio y 1.13 Diseño de plantillas con Thymeleaf"***

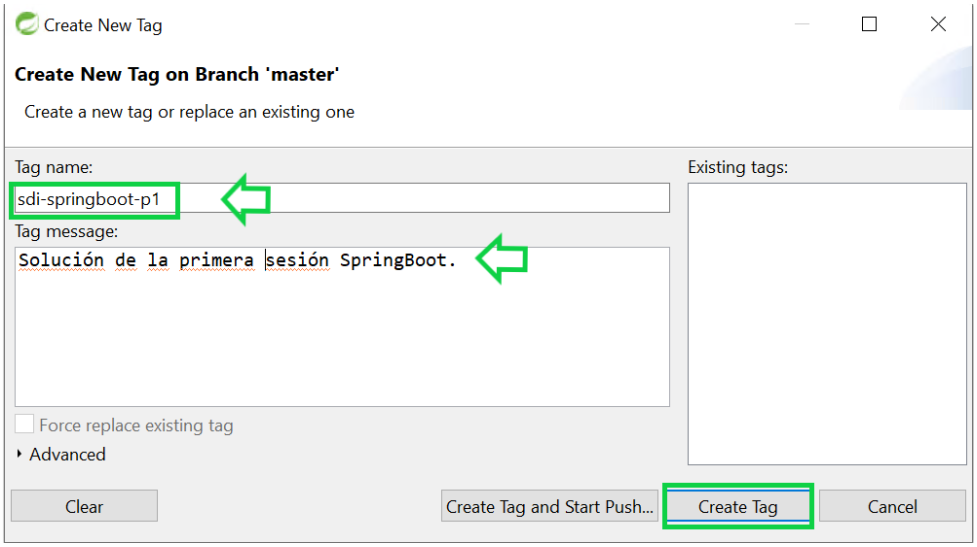
1.13 Etiquetar proyecto en GitHub

Nota: Etiquetar el proyecto en este punto con la siguiente etiqueta -> **sdi-springboot-p1.**

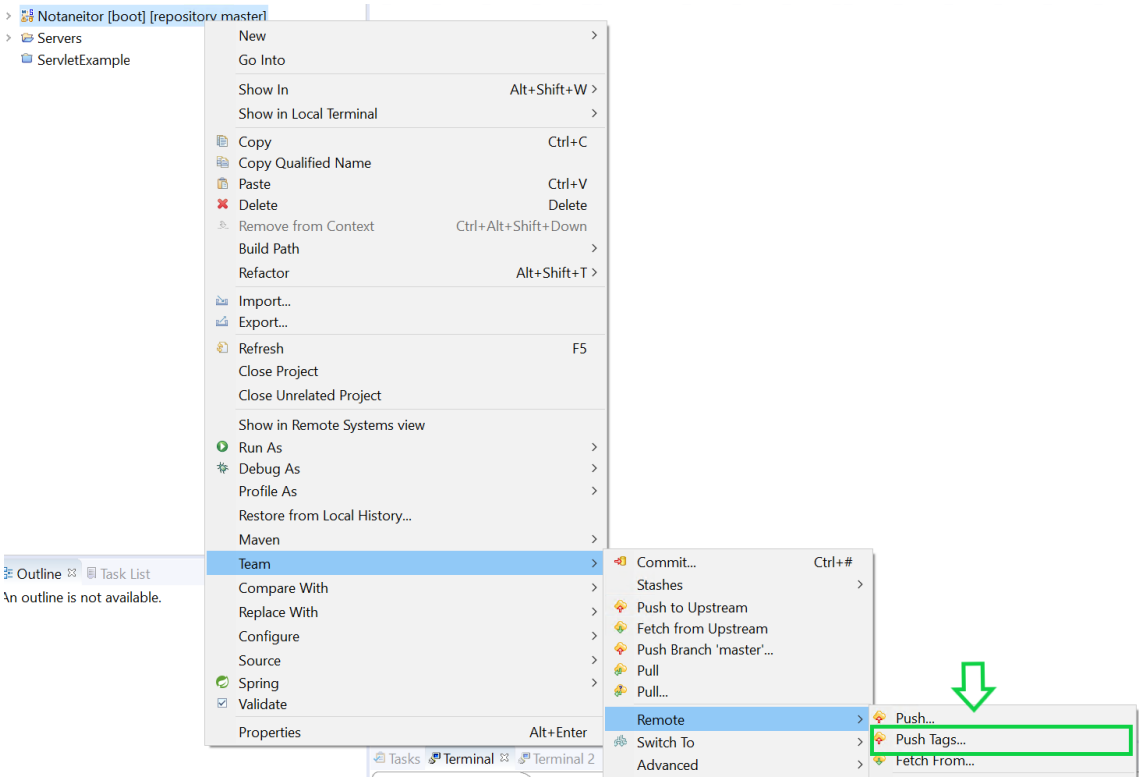
Para crear una **Release** en un proyecto que está almacenado en un repositorio de control de versiones lo común es utilizar etiquetas (Tags), que nos permitirá marcar el avance de un proyecto en un punto dado (una funcionalidad, una versión del proyecto, etc). Para crear una etiqueta en el proyecto en GitHub lo primero que vamos a hacer es hacer click derecho sobre el proyecto en STS e ir a la opción de **Team** -> **Advance -> Tag** que nos llevará hasta la ventana de creación de etiquetas(Tag).



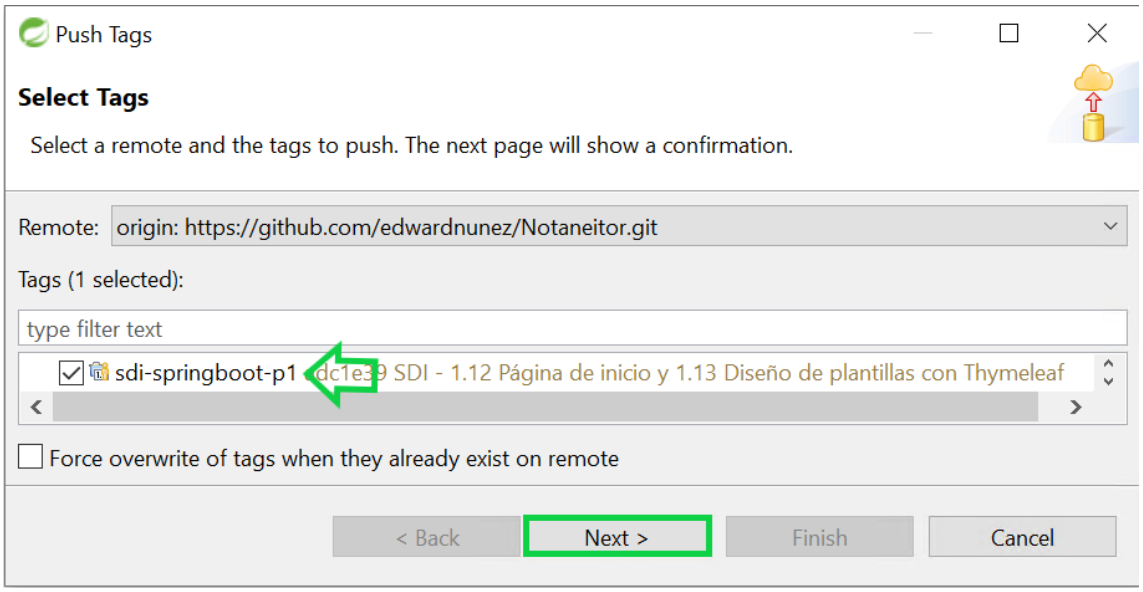
En la siguiente ventana indicamos el nombre de la etiqueta(**sdi-springboot-p1)** y un mensaje que indique la funcionalidad que se está etiquetado en este punto, luego pulsamos el botón **Create Tag.**



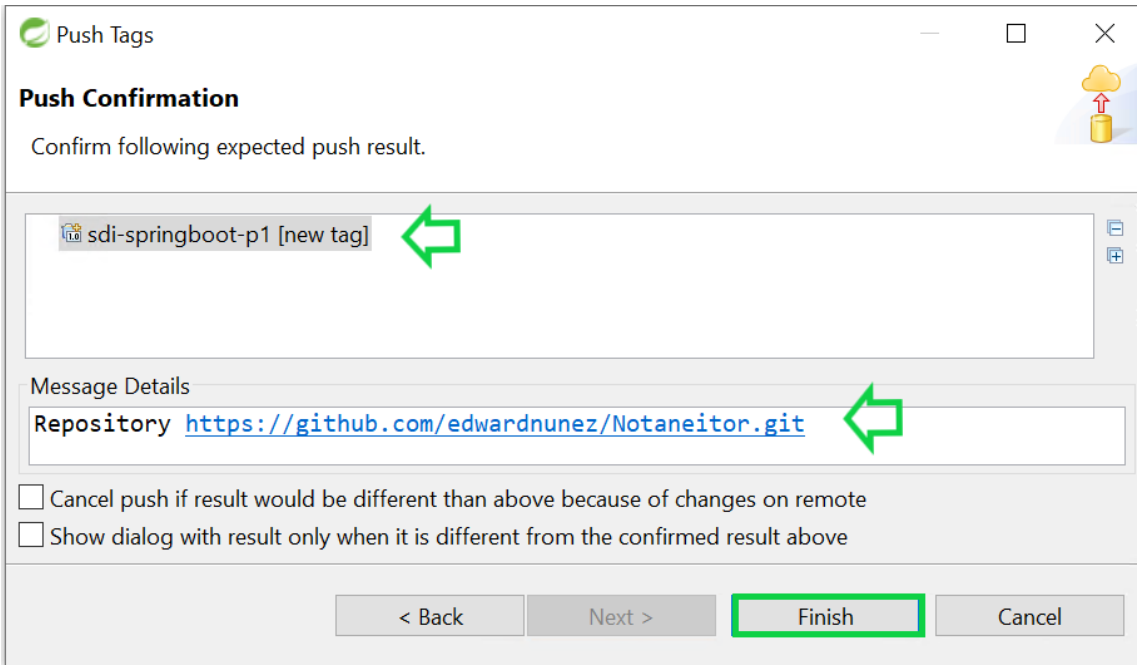
Con el paso anterior hemos creado la etiqueta en el repositorio local. Ahora tenemos que subir esa etiqueta al repositorio remoto(GitHub). Para subir una etiqueta al repositorio de Github hacemos click derecho sobre el proyecto e ir a la opción de **Team** -> **Remote -> Push Tag**.



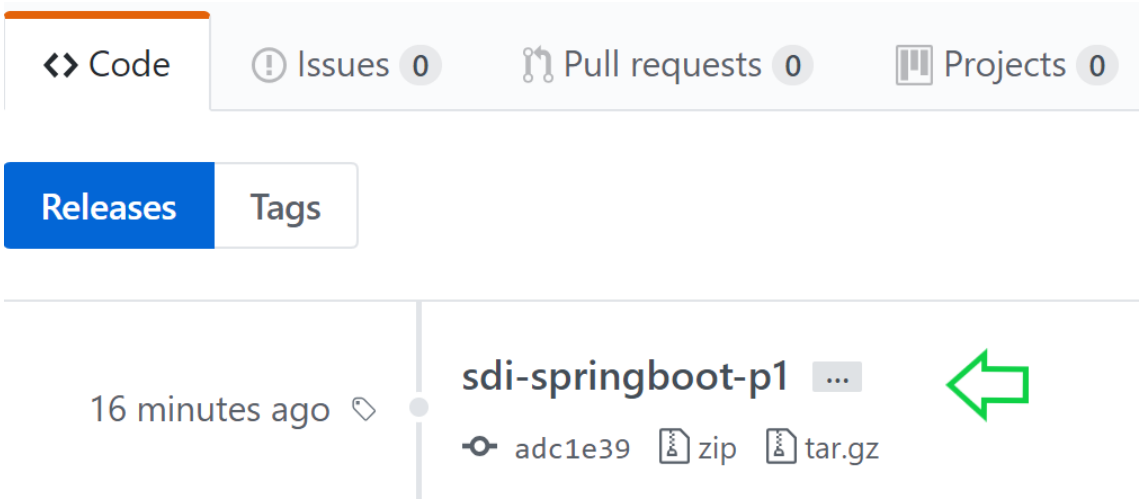
Una vez en la siguiente ventana seleccionamos las etiquetas que queremos subir al repositorio remoto y pulsamos el botón **Next**.



Una vez selecionadas las etiquetas (en nuestro caso sólo una) nos mostrará un mensaje con las etiquetas que estamos creando o modificanos y el repositorio al cuál estamos subiendo las etiquetas. Pulsamos el botón **Finish** para completar la tarea.



Una vez subida las etiquetas al repositorio, podemos verificar dichas etiquetas en nuestra cuenta y repositorio de GitHub. Vamos al repositorio correspondiento y al menú release y nos mostrára las etiquetas creada en el repositorio.



1.14 Resultado esperado en el repositorio de GitHub

Al revisar el repositorio de código en GitHub, los resultados de los commits realizados deberían ser parecidos a estos.