



# Sistemas Distribuidos e Internet

## Arquitectura MVC con Spring Boot

### Sesión 2

### Curso 2022/2023

<b>1</b>	<b>INTRODUCCIÓN AL DESARROLLO DE APLICACIONES WEB BASADAS EN MVC CON SPRING BOOT.....</b>	<b>2</b>
1.1	CONFIGURACIÓN DEL ENTORNO DE DESARROLLO.....	2
1.2	SPRING.....	3
1.3	CREACIÓN DE PROYECTO Y DESPLIEGUE .....	3
1.4	CONFIGURAR EL REPOSITORIO GIT Y CREACIÓN DEL PRIMER COMMIT .....	6
1.5	FICHERO DE CONFIGURACIÓN Y DESPLIEGUE DE LA APLICACIÓN .....	7
1.6	CODIFICACIÓN DEL WORKSPACE UTF-8 .....	10
1.7	GESTIÓN DE DEPENDENCIAS CON LIBRERÍAS .....	10
1.8	ARQUITECTURA MVC .....	12
1.9	CONTROLADOR.....	13
1.10	BEANS - SERVICIOS, INYECCIÓN DE DEPENDENCIAS .....	21
1.11	MODELO - ACCESO A DATOS SIMPLE.....	25
1.12	VISTAS – MOTOR DE PLANTILLAS THYMELEAF.....	29
1.13	PÁGINA DE INICIO .....	38
1.14	DISEÑO DE PLANTILLAS CON THYMELEAF .....	39
1.15	RESULTADO ESPERADO EN EL REPOSITORIO DE GITHUB .....	44



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

En esta práctica veremos una introducción al desarrollo de aplicaciones web basadas en el patrón MVC con Spring Boot.

## 1.1 Configuración del entorno de desarrollo

Para desarrollar aplicaciones web en Java es necesario trabajar con un Entorno de Desarrollo Integrado (IDE) adecuado. Uno de los IDE más utilizados en entornos empresariales es **IntelliJ IDEA**<sup>1</sup>. También es buena opción el IDE **Spring Tool Suite (STS)**<sup>2</sup>, herramienta que está basada en el IDE Eclipse. Ambos entornos incluyen todo lo necesario para facilitarnos el desarrollo de aplicaciones web en Java y también están optimizados para el uso del framework Spring.

### Prerrequisitos / Software utilizado:

1. **IntelliJ IDEA ULTIMATE** versión 2021.3.2<sup>3</sup>: Entorno de Desarrollo Integrado.
2. **JDK-17**: Java Development Kit.
3. **Git 2.20.1**: Sistema de control de versiones.
4. **Apache Tomcat Web Server** versión 9.0.41: Contendor de Servlet open source.
5. **Bootstrap** versión 4.5.2: Framework para crear sitios web responsive.
6. **jQuery** versión 3.5.1: Biblioteca multiplataforma de JavaScript.
7. **HSQLDB** versión 2.6.1: Motor de base de datos relacional.

### Notas:

1. Inicialmente vamos a crear el proyecto inicial en IntelliJ IDEA y luego lo sincronizaremos con un repositorio remoto en GitHub.
2. De manera progresiva, el alumno deberá ir subiendo el código a GitHub en los puntos de control especificados en el documento.

<sup>1</sup> <https://www.jetbrains.com/idea/download/>

<sup>2</sup> <https://spring.io/tools>

<sup>3</sup> En MACOS a 11 de enero de 2022 la versión disponible es 2021.3.1. Es igualmente válida.



## 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<sup>4</sup>, inversión de dependencias<sup>5</sup> e inyección de dependencias<sup>6</sup>, acceso a datos, gestión de transacciones, acceso remoto, mensajería, administración remota, autenticación y autorización, etc.

### Spring Boot<sup>7</sup>

Es un proyecto (o facilitador de) Spring que utiliza las características del framework combinadas con algunas nuevas creadas específicamente para conseguir desarrollos y despliegues **más ágiles**, tales como:

- 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 o Gradle.
- 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 pueden utilizar los lenguajes de programación Kotlin y Groovy, que se ejecutan sobre la máquina Virtual de Java.

Las dos alternativas más habituales para crear un nuevo proyecto Spring boot son:

1. Crear un proyecto Maven desde el entorno de desarrollo e incluir la dependencia en el pom.xml (Maven).

---

<sup>4</sup> El framework invoca al código ante ciertos eventos, tales como pulsar un botón, etc.

<sup>5</sup> Según Martin Fowler, 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).

<sup>6</sup> 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.

<sup>7</sup> <https://projects.spring.io/spring-boot/>



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)

Vamos a optar por la opción 2 por ser más directa, aunque todas son igual de válidas.

### !!!!MUY IMPORTANTE!!!!

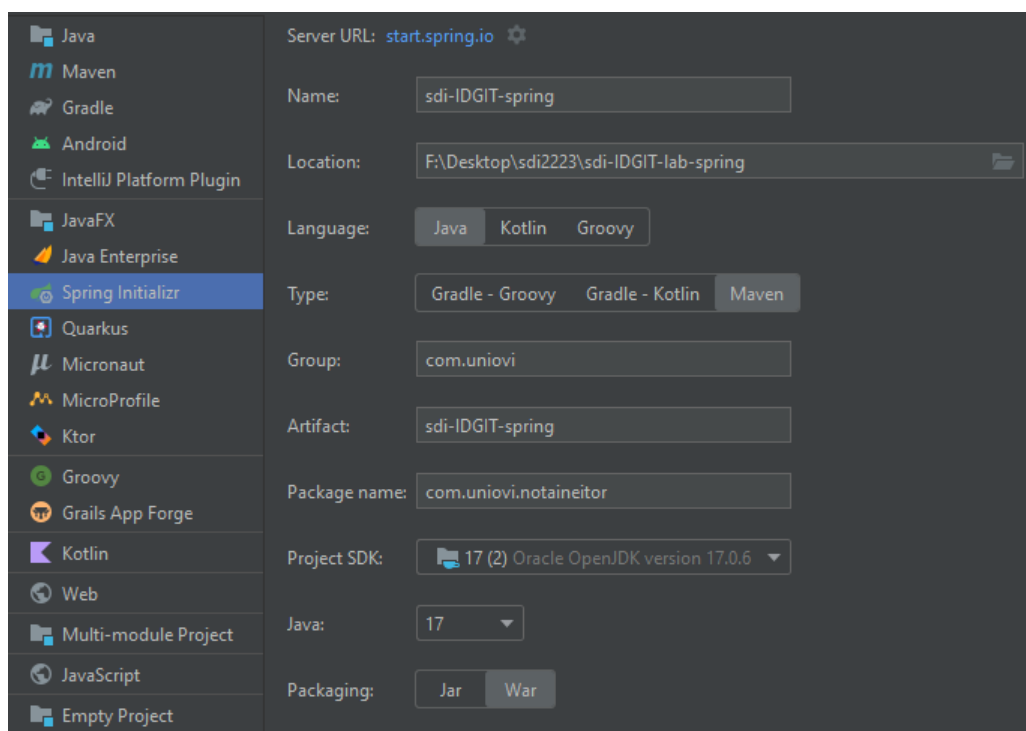
Cada alumno deberá usar como nombre de repositorio **sdi-IDGIT-lab-spring** y como proyecto **sdi-IDGIT-spring**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI2223.pdf del CV.

En resumen, por ejemplo, para el alumno **IDGIT=2223-101**:

Repositorio remoto:	<b>sdi-2223-101-lab-spring</b> (repositorio remoto nuevo)
Repositorio local:	<b>sdi-2223-101-lab-spring</b> (repositorio local nuevo)
Nombre proyecto:	<b>sdi-2223-101-spring</b> (sin “lab” por acortar el nombre)
Colaborador invitado:	<b>sdigithubuniovi</b> (hay que invitarlo al nuevo repositorio)

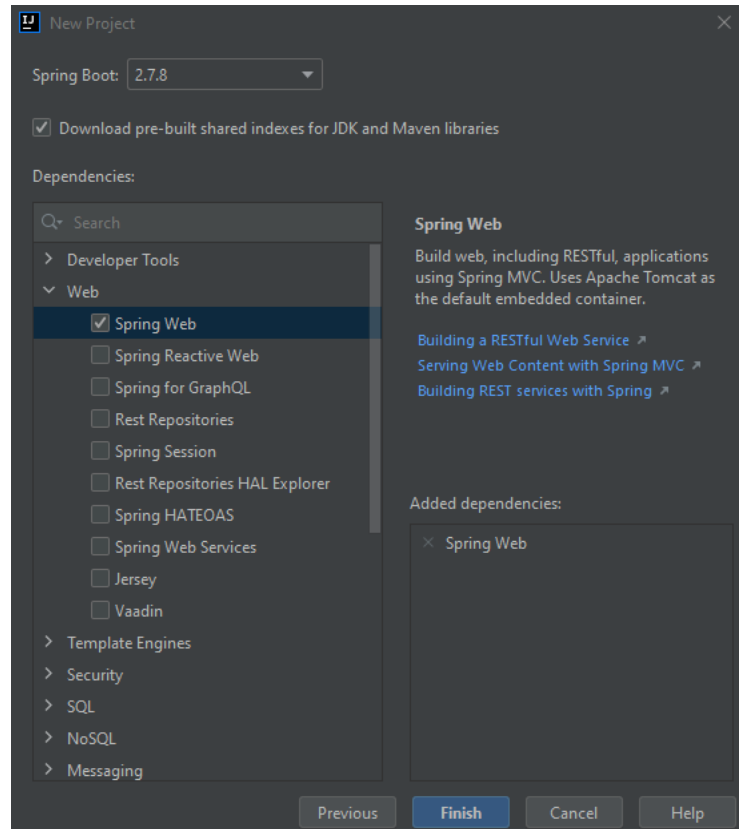
Para crear un nuevo proyecto Spring Boot usando IntelliJ IDEA, vamos al menú **File|New|Project**, seleccionamos la opción **Spring Initializr**, y luego los datos de proyecto, incluyendo la versión del SDK que vamos a utilizar.

En el enunciado denominaré al artefacto “notaneitor”. Sin embargo, debes ajustar el nombre a tu IDGIT: **sdi-2223-101-spring**. El artefacto servirá para construir el nombre del paquete principal del proyecto. Ajustamos los parámetros como en la imagen y pulsamos **Next**.



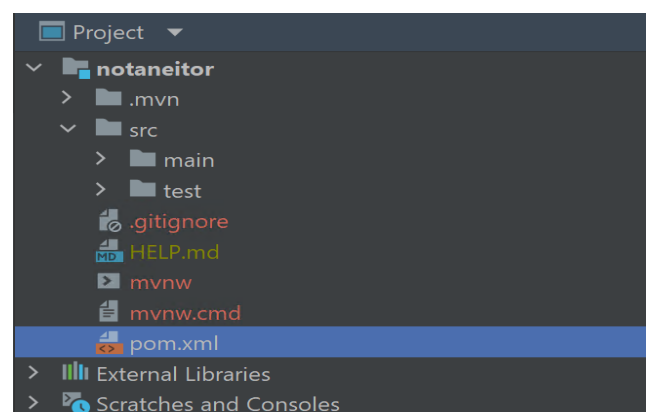


Pulsamos en el botón **Next** y seleccionamos la **versión 2.7.8 de Spring Boot** y la **dependencia Web → Spring Web** en el asistente. Estas son las dependencias con las que se iniciará el proyecto, pero en el futuro podemos añadir más si las necesitamos. Ver la siguiente imagen:



Finalmente, pulsamos el botón **Finish** para finalizar la creación de la plantilla del proyecto.

El proyecto en sí tiene un contenido mínimo, básicamente: una estructura de carpetas, una clase principal y varios ficheros de configuración. Una vez se descarguen todas las dependencias del proyecto, veremos la estructura de carpetas tal como se muestra en la siguiente imagen:



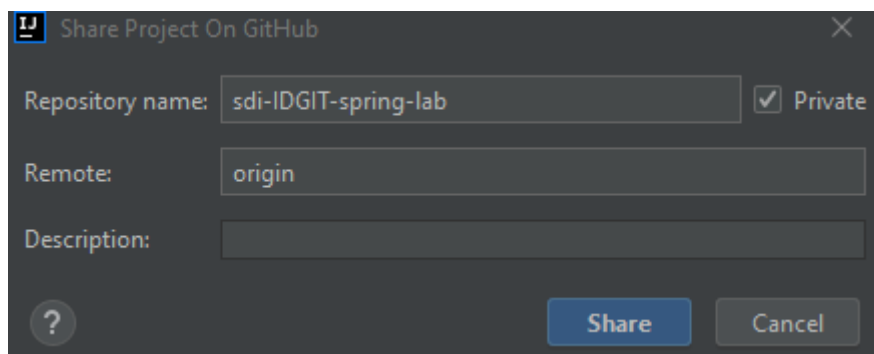


## 1.4 Configurar el repositorio Git y creación del primer commit

Dado que este proyecto va a estar vinculado a un repositorio remoto nuevo (sdi-IDGIT-lab-spring) tenemos que ir a **VCS → Share Project on Github** para vincularlo a un nuevo repositorio.

Tal y como se muestra en la imagen inferior, se establece el nombre del repositorio. Además, marcaremos el repositorio como privado. Seguidamente, pulsamos **Share**.

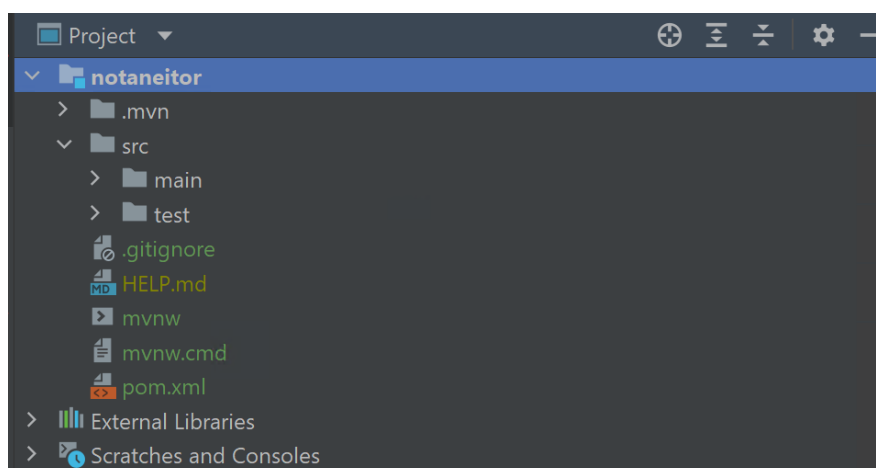
No olvides posteriormente invitar como colaborador a **sdigithubuniovi**.



Si no está configurado la cuenta de GitHub en el IDE, recuerda que en el guion anterior se explicó el procedimiento.

### *Commit inicial del proyecto*

Para realizar el primero commit del nuevo proyecto, primero nos colocamos sobre la carpeta del proyecto y hacemos clic derecho y luego vamos a la opción **Git|Add** para añadir los nuevos cambios al **repositorio LOCAL**. Al añadir los nuevos ficheros al index, estos cambian a color verde, como se muestra en la siguiente imagen:



A continuación, hay que hacer **commit y push** para subir los cambios a ambos repositorios (local y remoto). Para esto, hacemos **clic derecho sobre la carpeta del proyecto**, opción **Git|Commit Directory**, verificamos los cambios añadidos, escribimos el **mensaje del commit** y finalmente presionamos el botón **Commit and Push**.



**Nota: Incluir el siguiente Commit Message ->**

**“SDI-IDGIT-2.1-Create Spring project.”**

**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**

**“SDI-2223-101-2.1-Create Spring project.”**

**(No olvides incluir los guiones y NO incluyas BLANCOS ni comillas)**

## 1.5 Fichero de configuración y despliegue de la aplicación

Uno de los ficheros más interesantes es el **pom.xml** (Maven gestiona las librerías del proyecto). Abrimos el fichero **pom.xml** y vemos su estructura.

```
m pom.xml (notaneitor) x
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.6.3</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.uniovi</groupId>
12  <artifactId>notaneitor</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <packaging>war</packaging>
15  <name>notaneitor</name>
16  <description>notaneitor</description>
17  <properties>
18    <java.version>11</java.version>
19  </properties>
20  <dependencies>
21    <dependency>
22      <groupId>org.springframework.boot</groupId>
23      <artifactId>spring-boot-starter-web</artifactId>
24    </dependency>
```

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

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

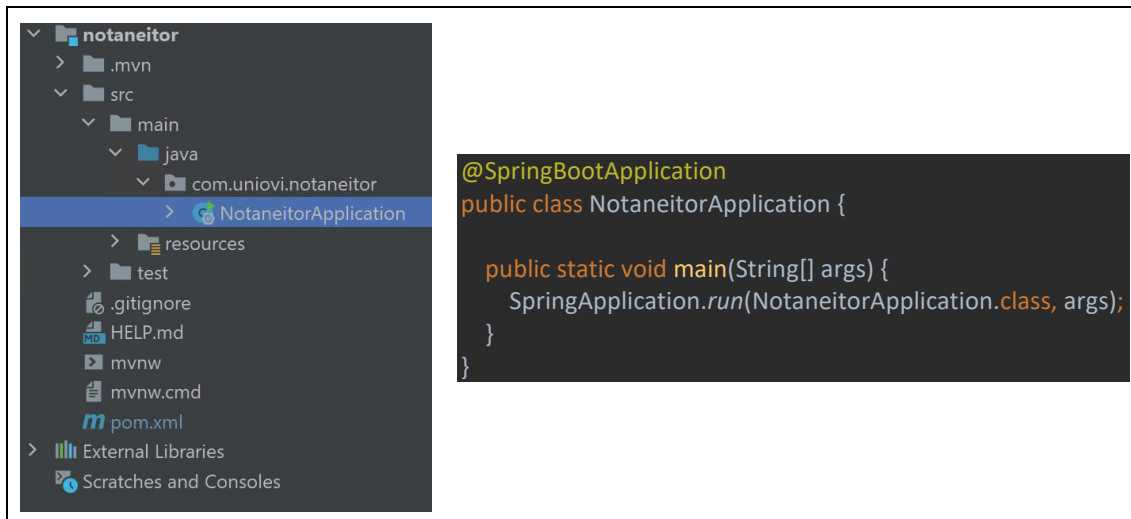
De momento no vamos a modificar ninguno, pero casi todas las futuras modificaciones en el fichero POM se van a centrar en agregar elementos al **bloque <dependencies>**. Esta es la forma de agregar librerías a un proyecto utilizando Maven, al incluir la librería en la sección <dependencies> esta se agrega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



```
<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/** está la clase principal del proyecto, **NotaneitorApplication.java**. Se trata de una clase Java normal, con la anotación **@SpringBootApplication** y un método **main**.

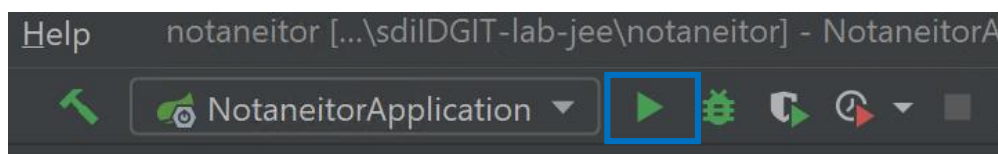


The screenshot shows an IDE interface. On the left, the project structure is visible, with the file **NotaneitorApplication** selected under the path **src/main/java/com.uniovi.notaneitor**. On the right, the code for **NotaneitorApplication** is displayed:

```
@SpringBootApplication
public class NotaneitorApplication {

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

Esta es la clase principal de la aplicación. La seleccionamos y ejecutamos el proyecto con la opción “Run/Play” (triángulo verde) de la barra superior.



Veremos que al ejecutar la aplicación se empieza a mostrar información en la ventana **Console** del IDE. Las aplicaciones Spring Boot son “Standalone applications”, al ejecutarlas despliegan su propio servidor (Es este caso en un Tomcat y en el puerto 8080).

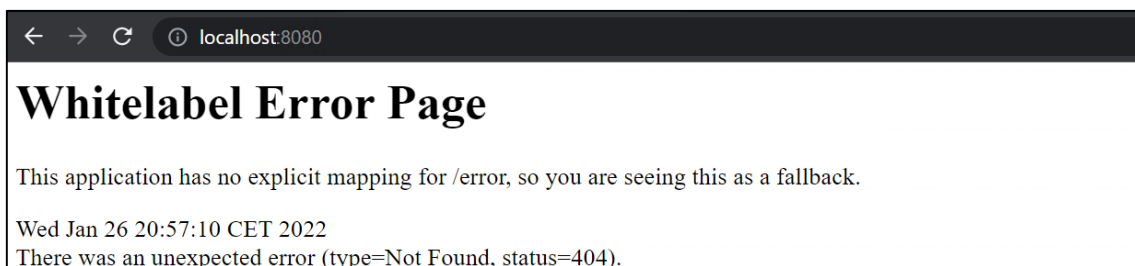




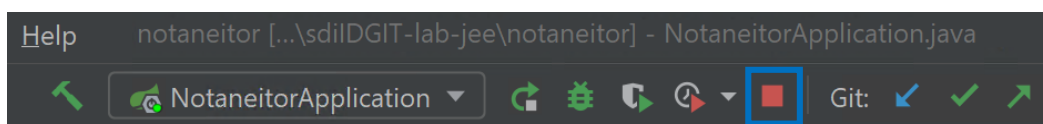
```
Run: NotaneitorApplication x
Console
Endpoints
: Tomcat initialized with port(s): 8080 (http)
: Starting service [Tomcat]
: Starting Servlet engine: [Apache Tomcat/9.0.41]
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 3359 ms
: Initializing ExecutorService 'applicationTaskExecutor'
: Tomcat started on port(s): 8080 (http) with context path ''
: Started NotaneitorApplication in 4.945 seconds (JVM running for 7.446)
```

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



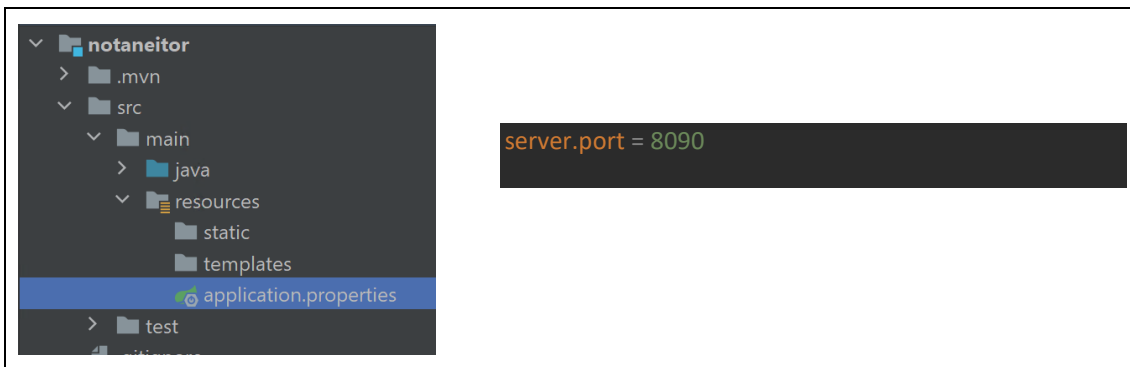
Para detener la aplicación pulsamos en el botón **Stop**. Debemos recordar que hay que parar y volver a desplegar la aplicación cada vez que realicemos cambios.



### ***Cambiar el puerto de escucha del servidor***

Si el puerto 8080 del equipo está ocupado por otro proceso podemos cambiarlo, para esto, abrimos el fichero de propiedades situado en **src/main/resources -> application.properties**.

Al abrir el fichero podemos observar que por defecto está vacío, eso se debe a que no hace falta especificar ninguna propiedad puesto que todas tienen valores por defecto. Este framework sigue la filosofía de “Convención sobre configuración”, todas las propiedades usan valores por defecto, únicamente las definimos si queremos modificar el valor establecido por convención. Para que la aplicación escuche por el puerto 8090, incluimos la propiedad **server.port** con el valor tal como se muestra en esta imagen:

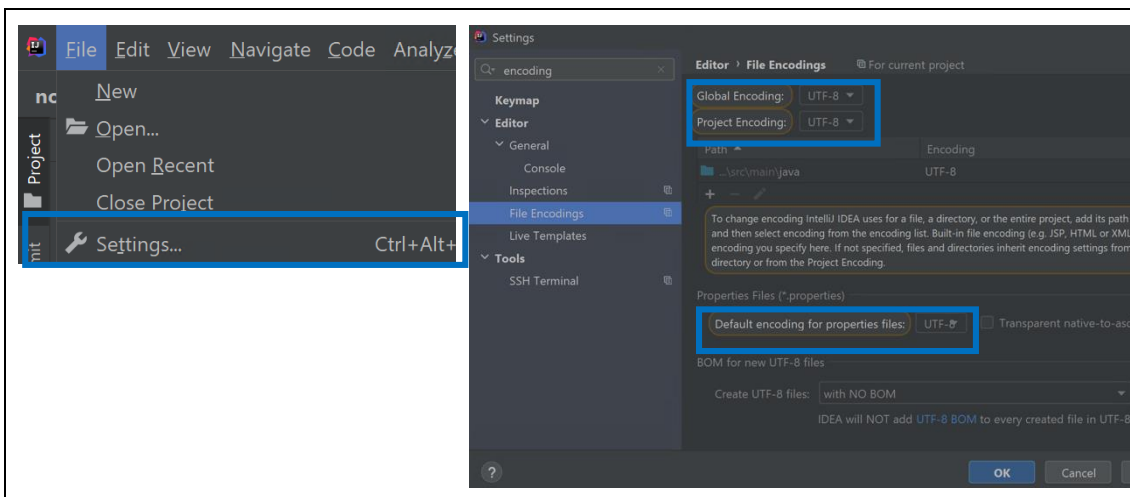


Para que los cambios tengan efectos debemos reiniciar la aplicación.

**Nota:** A partir de este punto en el guion se desplegará la aplicación en el **puerto 8090**.

## 1.6 Codificación del workspace UTF-8

Para ahorrarnos futuros problemas de codificación es buena idea asegurarnos de que la codificación del IntelliJ sea UTF-8. Para verificar esto vamos al menú **File|Settings** del IDE. Luego desde el menú de la izquierda seleccionamos **Editor|File Encodings** y modificamos la propiedad **Global encoding**, **Project Encoding** y **default encoding for properties files** al valor a **UTF-8**, en el caso que tengamos otros valores. La configuración quedará como se muestra en la siguiente imagen y finalmente pulsamos el botón OK:



## 1.7 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 para muchos aspectos importantes: la seguridad, gestión de datos, etc.

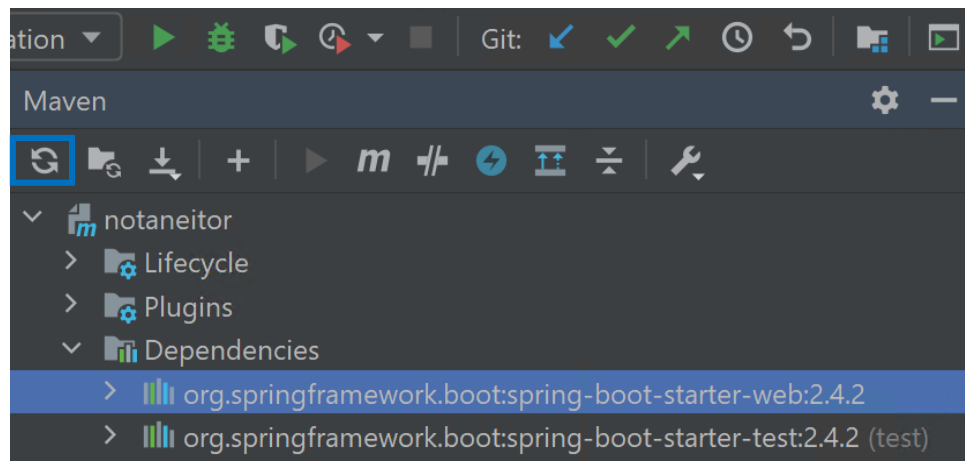
Las librerías incluyen muchas funcionalidades extras que pueden ser usada dentro de la aplicación. Todas las librerías de nuestra aplicación se gestionan a través de Maven, (no vamos a importar ningún .jar en el proyecto).



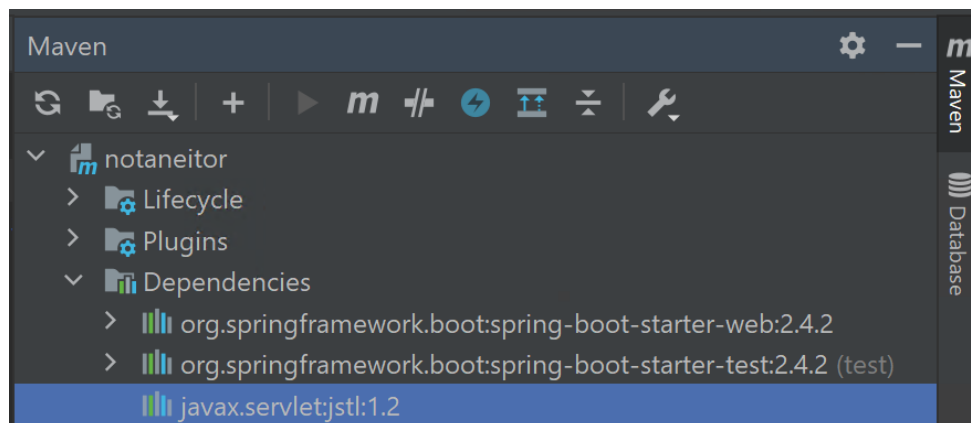
A modo de prueba, vamos a comenzar importando una dependencia para poder utilizar la librería de JSTL (JSP - Standard Tag Library) en la aplicación. Incluimos la siguiente etiqueta dentro del bloque **<dependencies>**.

```
<dependencies>
...
<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>
```

Para descargar la librería JSTL vamos al panel de Maven (Panel lateral derecho de IntelliJ) y pulsamos el botón Reload all maven Project, tal como se muestra en la siguiente imagen:



Al actualizar las dependencias aparecerá la librería **jstl 1.2** que acabamos de incluir en el proyecto, tal como se muestra en la siguiente imagen:





## 1.8 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 al desarrollo:

- Mejorar la arquitectura y la robustez.
- Fuerza a utilizar una arquitectura modular.
- Fomenta la separación entre capas.
- Favorece la reutilización de código.
- En general, esta arquitectura favorece el mantenimiento y la extensión de la aplicación en comparación con arquitecturas basadas en otros patrones.

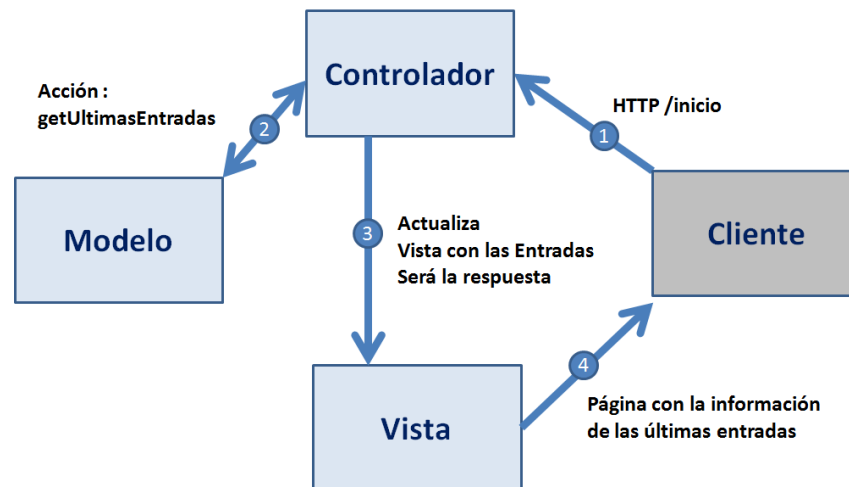
Esta arquitectura se divide en tres capas claramente diferenciadas: el **Modelo**, la **Vista** y el **Controlador**.

**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 la 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 y, con un pequeño script, recorre la lista de Entradas que le ha sido pasada como parámetro generando 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.

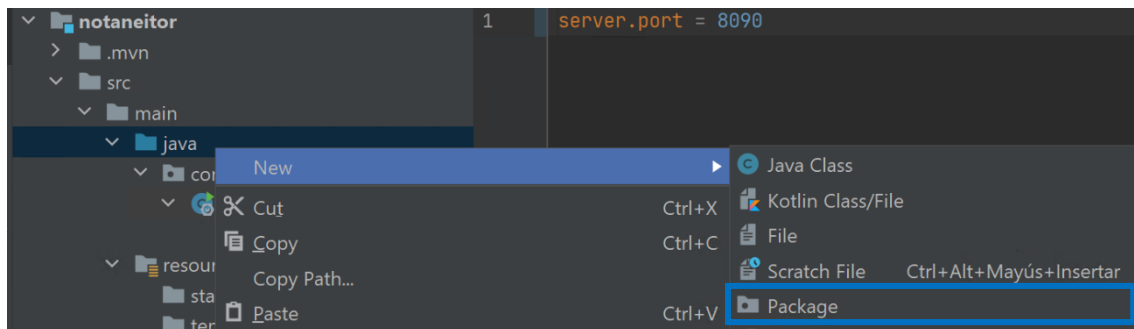
## 1.9 Controlador

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

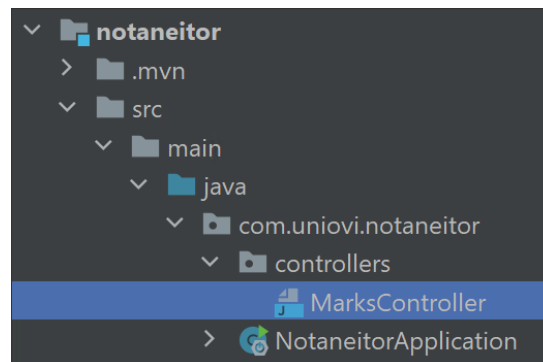
Una aplicación web suele tener al menos un controlador.

A continuación, vamos a crear un nuevo paquete Java `com.uniovi.notaneitor.controllers` y dentro del paquete la clase Java: **MarksController.java**.

Hacemos clic derecho sobre la carpeta **Java | New | Package**, nos aparecerá una ventana donde escribiremos el nombre del paquete y pulsaremos Enter.



Para crear la clase se sigue el mismo procedimiento anterior, pero colocándonos encima del controlador. Hacemos clic derecho sobre **controllers** | **New** | **Java Class**, luego ponemos el nombre de la clase (MarksController) y pulsamos sobre la opción Class.



Debemos incluir la anotación **@RestController** en la clase, esto indica que la clase es un controlador (Rest) que responde a peticiones Rest (No nos preocupemos por el concepto Rest, ya que lo veremos más adelante). Inicialmente, puede que nos muestre 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 o simplemente autocompletar el código pulsando las teclas CTRL + SPACE y, seleccionando la anotación correspondiente, se importará automáticamente el paquete:

```
package com.uniovi.notaneitor.controllers;  
  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class MarksController {  
}
```

Ahora, incluimos un método 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.

```
package com.uniovi.notaneitor.controllers;  
  
import org.springframework.web.bind.annotation.RequestMapping;
```



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

@RestController
public class MarksController {

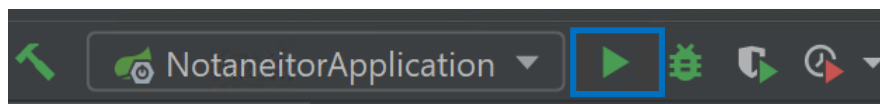
    @RequestMapping("/mark/list")
    public String getList() {
        return "Getting List";
    }

    @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 correcto es crear diferentes controladores, por ejemplo, uno para notas, otro para asignaturas, otro para alumnos, etc.

## ***Peticiones Web***

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

## ***Peticiones GET***

Las peticiones GET pueden incluir parámetros solamente en la URL. Las peticiones POST los incluyen en el cuerpo (body) de la petición HTTP.





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

**Nota:** Si escribimos la anotación en lugar de copiarla, IntelliJ nos importará automáticamente el paquete correspondiente.

```
@RequestMapping("/mark/details")
public String getDetail(@RequestParam Long id) {
    return "Getting Details => " + id;
}
```

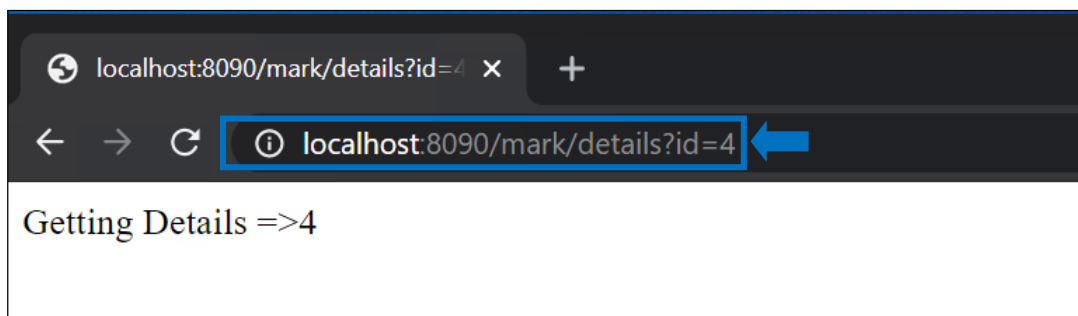
Tal y como lo hemos declarado la petición (Request), puede contener un parámetro con clave "id" (El controlador busca un parámetro en cualquier parte de la URL con clave id).

**A continuación, 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>



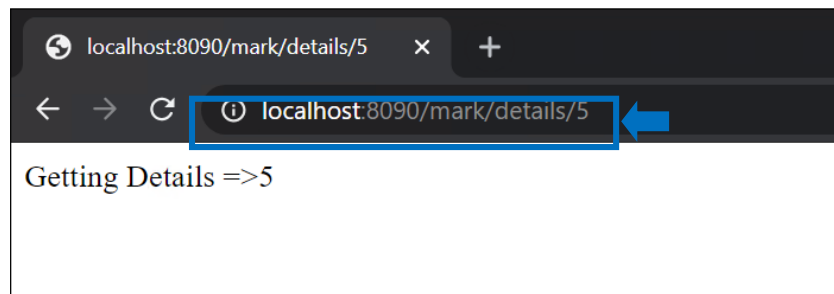
Otra forma común de incluir parámetros GET es en el propio Path, **sin tener que indicar la clave del parámetro, porque la clave viene dada por su posición en el Path.**

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 Details => " + id;
}
```

Probamos que este enfoque también funciona de forma correcta.  
<http://localhost:8090/mark/details/5/>





## Peticiones POST

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 importando 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.

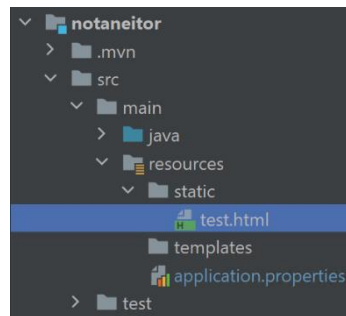
Si los parámetros van a ser enviados a través de un formulario cada uno va a tener una **clave** y un **valor**. No obstante, el cuerpo de un body puede estar codificado en diversos formatos: “json”, “application/x-www-form-urlencoded”, “text/plain”, para nosotros esto es poco relevante, ya que el framework es capaz de extraer los datos 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** se utiliza para guardar recursos estáticos (normalmente fotografías, css, js) a estos recursos se puede acceder sin necesidad de pasar por el controlador.

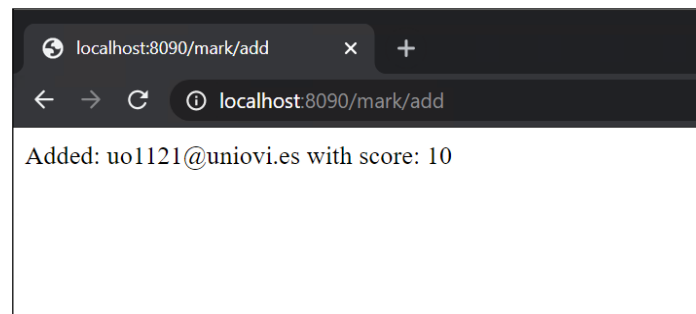


Implementamos un formulario simple que envíe mediante **POST** a la URL **/add** los parámetros con las claves **email** y **score**.

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

Detenemos la aplicación y la volvemos a ejecutar, accedemos al recurso estático <http://localhost:8090/test.html>. A continuación, introducimos unos datos de ejemplo.

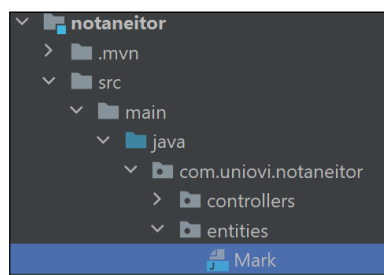
Al pulsar "Send", debería mostrarnos la respuesta del controlador a **POST /mark/add**.



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.notaneitor.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 IntelliJ).



Para usar el asistente nos colocamos dentro de la clase java (Mark.java) y luego vamos a la opción *Code|Generate|Getters and Setters* y luego seleccionar los atributos para los cuales queremos generar estos métodos. La implementación será la siguiente:

```
package com.uniovi.notaneitor.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 Mark, hay que importar el paquete *com.uniovi.notaneitor.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();
}
```

Si el objeto del modelo en el que se mapean los parámetros tiene más atributos de los recibidos estos atributos no se inicializarán.

En este caso observaremos ese problema con el atributo id.

A continuación, desplegamos la aplicación de nuevo y probamos.

Tests Page

localhost:8090/test.html

Descripción:

Puntuación:

**Nota: Incluir el siguiente Commit Message ->**  
**“SDI-IDGIT-2.2-Trabajando con Controladores.”**  
**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**  
**“SDI-2021-101-2.2-Trabajando con Controladores.”**  
**(No olvides incluir los guiones y NO incluyas BLANCOS)**



## 1.10 Beans - Servicios, inyección de dependencias

### Crear un Bean

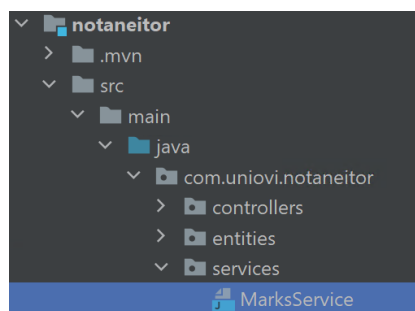
En primer lugar, vamos a ampliar la implementación de la clase `Mark(Nota)`, añadimos un **constructor con parámetros y un constructor sin parámetros** (al incluir el “constructor con parámetros” desaparece el constructor implícito sin parámetros y debemos incluirlo explícitamente). Implementamos también el método `toString()`. En ambos casos podemos utilizar el asistente de IntelliJ para generar el código correspondiente. Para usar el asistente nos colocamos dentro de la clase Java (`Mark.java`) y luego vamos a la opción `Code | Generate | Constructor`. Luego hacemos los mismos pasos para el método: `Code | Generate | toString()`. La siguiente imagen muestra el código generado.

```
public Mark() {  
}  
  
public Mark(Long id, String description, Double score) {  
    this.id = id;  
    this.description = description;  
    this.score = score;  
}  
  
@Override  
public String toString() {  
    return "Mark{" + "id=" + id + ", description=" + description + "\n" + ", score=" + score + "};  
}
```

### Implementar un servicio

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 utilizará 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.notaneitor.services**, y dentro, creamos la clase **MarksService.java**





La anotación **@Service** indica que esta clase es un servicio. Si queremos que una función actúe como “inicializador” debemos incluir la anotación **@PostConstruct**, (en este caso lo vamos a utilizar 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<>();

    @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 el auto-importador del IDE 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.notaneitor.entidades.Mark;
```



### *Injectar beans*

¿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 el 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**. Puedes consultar más información en el siguiente enlace: <https://www.baeldung.com/spring-annotations-resource-inject-autowire>

```
@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();
}

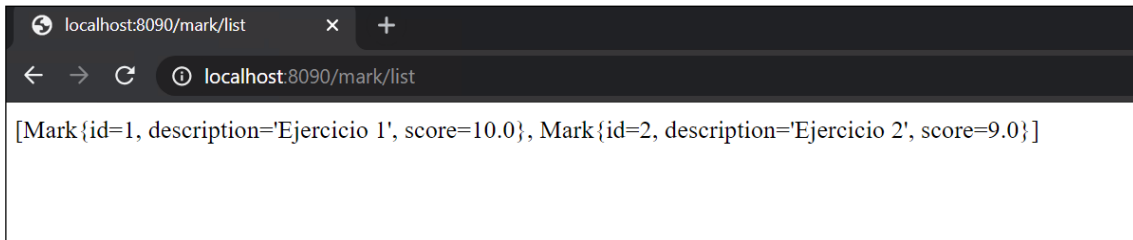
@RequestMapping("/mark/delete/{id}")
public String deleteMark(@PathVariable Long id) {
    marksService.deleteMark(id);
    return "Ok";
}
```

Reiniciamos la aplicación y comprobamos que continúa funcionando de la forma esperada. Podemos ver que se pueden **añadir, eliminar y listar** las notas.

- <http://localhost:8090/mark/list>

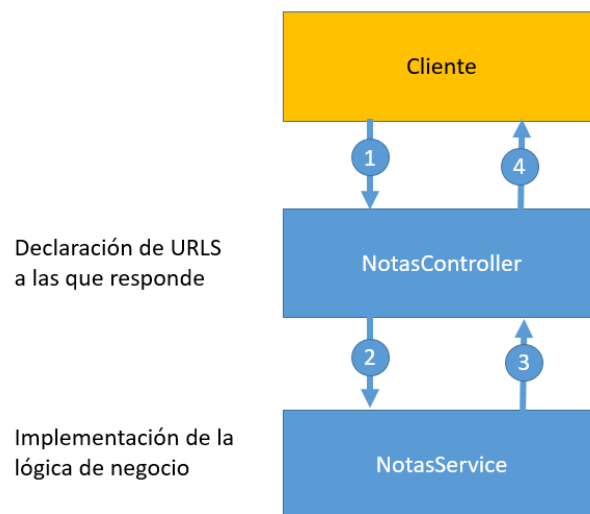


- <http://localhost:8090/mark/details/1>
- <http://localhost:8090/mark/delete/1>



El servicio se instancia una única vez en la creación de la aplicación. Esta instancia puede ser inyectada y utilizada en cualquier parte de la aplicación, empleando la anotación **@Autowired** en los componentes que consumirán la instancia.

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



**Nota: Incluir el siguiente Commit Message ->**

**“SDI-IDGIT-2.3-Beans, servicios e inyección de dependencias.”**

**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**

**“SDI-2021-101-2.3-Beans, servicios e inyección de dependencias.”**

**(No olvides incluir los guiones y NO incluyas BLANCOS)**





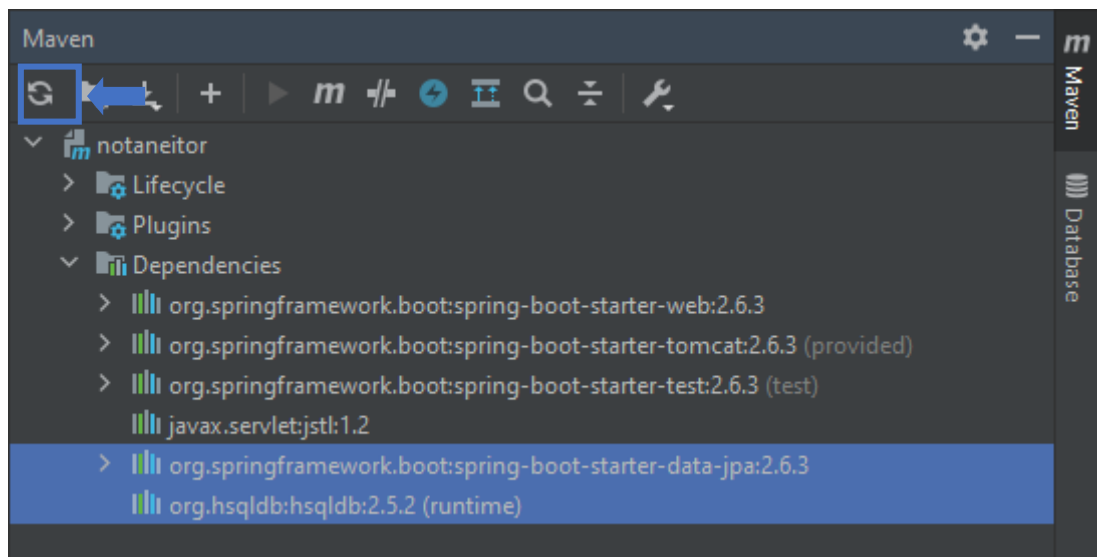
## 1.11 Modelo - Acceso a datos Simple

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.

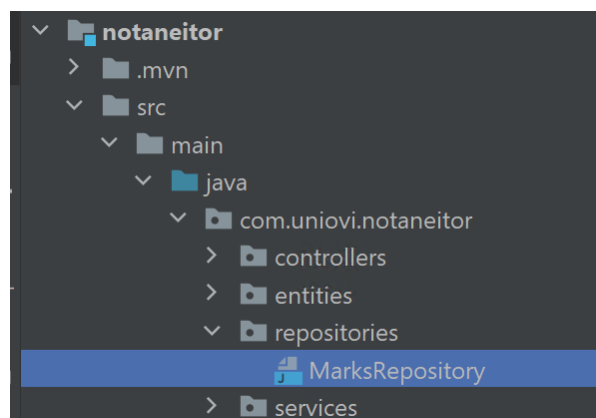
Agregamos al fichero **pom.xml** las dependencias de **JPA** y **HSQLDB**

```
<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 proyecto, desde el panel Maven en IntelliJ IDE.



Para crear la capa de modelo para persistir las notas, creamos el paquete **com.uniovi.notaneitor.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.

```
package com.uniovi.notaneitor.repositories;  
  
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 es ahí donde queremos utilizarlo.

Nos falta incluir las anotaciones de JPA en la clase **com.uniovi.notaneitor.entities.Mark** (Notas) para que sea reconocida por JPA como una entidad de repositorio. Indicamos que se mapeará 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 **MarksRepository**.

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

```
@Service  
public class MarksService {  
  
    @Autowired  
    private MarksRepository marksRepository;  
  
    /*  
    private List<Mark> marksList = new LinkedList<>();  
  
    @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() {  
    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 último + 1 de la lista  
    marksRepository.save(mark);  
}  
  
public void deleteMark(Long id) {  
    marksRepository.deleteById(id);  
}  
}
```

### *Desplegar el motor de base de datos*

Ahora solo nos queda desplegar un servidor de la base de datos HSQLDB en nuestra máquina 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\_2.7.1**

Descomprimos el fichero descargado y ejecutamos el fichero **hsqldb -> bin -> runServer.bat**.

La versión para MACOSX/Linux la puedes crear así:

```
$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
```



```
Windows PowerShell
PS D:\dev\Databases\hsqldb-2.4.1\hsqldb\bin> .\runServer.bat

D:\dev\Databases\hsqldb-2.4.1\hsqldb\bin>cd ..\data
[Server@4883b407]: Startup sequence initiated from main() method
[Server@4883b407]: Could not load properties from file
[Server@4883b407]: Using cli/default properties only
[Server@4883b407]: Initiating startup sequence...
[Server@4883b407]: Server socket opened successfully in 0 ms.
ene. 27, 2022 12:45:28 P. M. org.hsqldb.persist.Logger logInfoEvent
INFO: checkpointClose start
ene. 27, 2022 12:45:28 P. M. org.hsqldb.persist.Logger logInfoEvent
INFO: checkpointClose synched
ene. 27, 2022 12:45:28 P. M. org.hsqldb.persist.Logger logInfoEvent
INFO: checkpointClose script done
ene. 27, 2022 12:45:28 P. M. org.hsqldb.persist.Logger logInfoEvent
INFO: checkpointClose end
[Server@4883b407]: Database [index=0, id=0, db=file:test, alias=] opened successfully in 360 ms.
[Server@4883b407]: Startup sequence completed in 375 ms.
[Server@4883b407]: 2022-01-27 12:45:28.459 HSQldb server 2.4.1 is online on port 9001
[Server@4883b407]: To close normally, connect and execute SHUTDOWN SQL
[Server@4883b407]: From command line, use [Ctrl]+[C] to abort abruptly
```

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 **application.properties** de la aplicación e introducimos las propiedades de conexión:

```
notaneitor
├── .mvn
├── src
│   ├── main
│   │   ├── java
│   │   ├── resources
│   │   │   └── static
│   │   └── templates
│   └── application.properties
```

```
server.port = 8090

spring.datasource.url=jdbc:hsqldb:hsqldb://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 última 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: Incluir el siguiente Commit Message ->**

**“SDI-IDGIT-2.4-Modelo, acceso a datos simple.”**

**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**

**“SDI-2021-101-2.4-Modelo, acceso a datos simple.”**

**(No olvides incluir los guiones y NO incluyas BLANCOS)**

## 1.12 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 motores de plantillas más populares es THYMELEAF<sup>8</sup>.

¿Por qué usar un motor de plantillas?

- Sintaxis sencilla.
- No solo insertan datos y contiene estructuras de control básicas (if, for, expresiones lógicas, etc.), además, algunos poseen 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, evolucionado y potente 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>
```

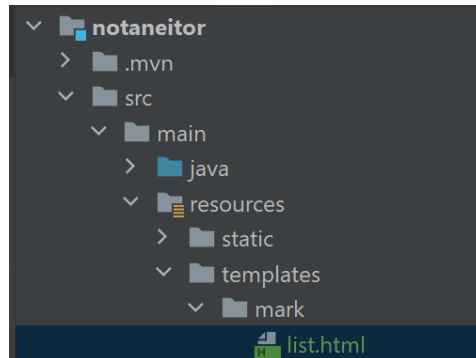
<sup>8</sup> <http://www.thymeleaf.org>



Después de agregar la dependencia debemos recargar los proyectos Maven desde el panel de Maven en IntelliJ.

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" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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 **<div>** con su correspondiente contenido.



- **th:text="{mark.id}":** inserta un nuevo texto en el componente HTML al que se asocia: dentro va a incluir el texto `<span> #Texto mark.id </span>`. 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 en 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";
    }
}
```

Además, debemos tener en cuenta es 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 = "/mark/add", method = RequestMethod.POST)
public String setMark(@ModelAttribute Mark mark) {
    marksService.addMark(mark);
    return "Ok";
}
```

La respuesta de este método 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 los métodos **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 reiniciamos la aplicación de nuevo. Probamos a añadir algunas notas desde la URL de prueba:** <http://localhost:8090/test.html>

Ahora 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 : ${markList}">
    <td scope="row" th:text="${mark.id}"> 1</td>
    <td th:text="${mark.description}"> Ejercicio 1</td>
    <td th:text="${mark.score}">10</td>
    <td><a th:href="${'/mark/details/' + mark.id}">detalles</a></td>
```

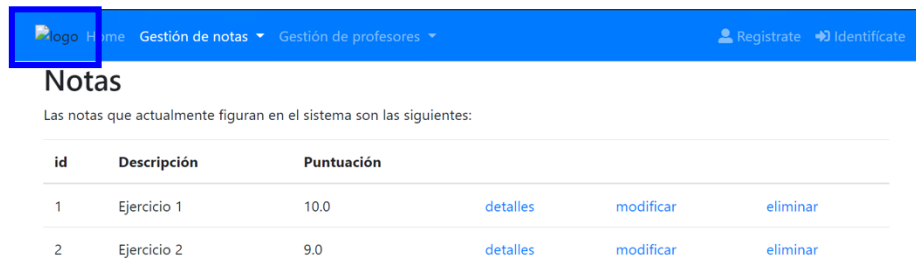




```
<td><a th:href='${'/mark/edit/' + mark.id}'>modificar</a></td>
<td><a th:href='${'/mark/delete/' + mark.id}'>eliminar</a></td>
</tr>
</tbody>
```

### *Incluir recursos de imágenes*

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



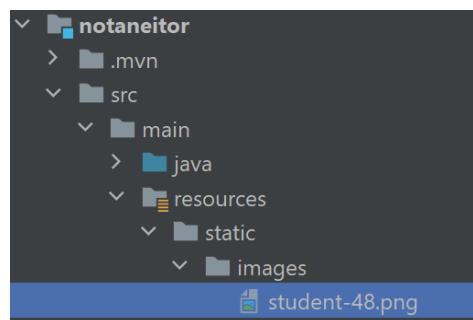
Según el código HTML la imagen debería estar en **images/student-48.png**

```

```

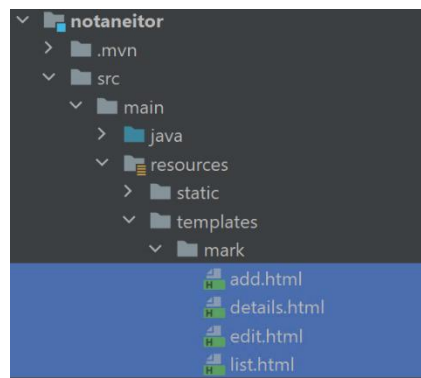
Debemos recordar que es mejor nombrar los recursos con paths absolutos.

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 **/images/** y copiar la fotografía.



### *Añadir las opciones de añadir, ver detalle y editar*

Para habitar las opciones de añadir, ver detalle y editar una nota, primero 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}**. Esta parte no se ha incluido en la plantilla, debemos abrir la plantilla **details.html** y modificarla

```
<div class="container" id="main-container">
  <h2>Detalles de la nota</h2>
  <div class="card">
    <div class="card-header">Descripción</div>
    <div class="card-body">
      <p class="card-text" th:text="${mark.description}>Ejercicio 1</p>
    </div>
  </div>
  <br>
  <div class="card">
    <div class="card-header">Puntuación</div>
    <div class="card-body">
      <p class="card-text" th:text="${mark.score}>100</p>
    </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>

 Home Gestión de notas ▼ Gestión de profesores ▼ Registrarse Identificarse

### Detalles de la nota

Descripción
Ejercicio 1
Puntuación
10.0

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 redirigir 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" id="main-container">
  <h2>Editar Nota</h2>
  <form class="form-horizontal" method="post" th:action="'${'/mark/edit/' + mark.id}'" >
    <div class="form-group">
      <label class="control-label col-sm-2" for="description">Descripción:</label>
      <div class="col-sm-10">
        <input th:value="'${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="score">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 envío 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(@ModelAttribute Mark mark, @PathVariable Long id){
    mark.setId(id);
    marksService.addMark(mark);
    return "redirect:/mark/details/"+id;
}
```

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



## Editar Nota

Descripción:

Ejercicio 1

Puntuación:

10,0

Modificar

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.

### Errores con el motor de plantilla Thymeleaf en IntelliJ

Al usar Spring Boot con Thymeleaf en IntelliJ IDEA, es posible que el IDE no reconozca la sintaxis del motor de plantillas. Por ejemplo, muestra las etiquetas en rojo, tal como se muestra en la siguiente imagen:

```
<div class="form-group">
  <label class="control-label col-sm-2" for="description">Descripción:</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>
```

Para ver los posibles errores, realizamos un análisis del código desde el menú **Code | Inspect Code...** y pulsando el botón **OK**. En la siguiente imagen se muestra como ejemplos algunos errores:

Messages: Code Analysis

- Warning:(8, 34) There is no locally stored library for the HTTP link.
- Warning:(10, 34) There is no locally stored library for the HTTP link.
- Warning:(13, 18) There is no locally stored library for the HTTP link.
- Warning:(14, 18) There is no locally stored library for the HTTP link.
- Error:(70, 49) Namespace 'th' is not bound
- Warning:(74, 18) Missing associated label
- Error:(74, 24) Namespace 'th' is not bound

Para resolver los errores anteriores tenemos que añadir el **namespace de thymeleaf** a todos los ficheros html que están utilizando el motor de plantilla. Añadimos las siguientes líneas a los ficheros add.html, details.html, edit.html y list.html:



```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<head>
```

**Nota: Incluir el siguiente Commit Message ->**  
**“SDI-IDGIT-2.5-Vistas, motor de plantillas Thymeleaf.”**  
**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**  
**“SDI-2021-101-2.5-Vistas, motor de plantillas Thymeleaf.”**  
**(No olvides incluir los guiones y NO incluyas BLANCOS)**

## 1.13 Página de inicio

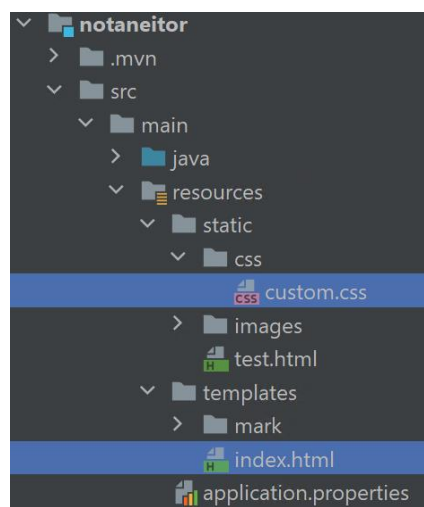
Actualmente si accedemos a la raíz de nuestra página web <http://localhost:8090> nos dará un error porque no tenemos configurada la página de inicio

Vamos a definir un nuevo controlador **HomeController** para gestionar la página de inicio, tal y como se muestra en el siguiente código:

```
package com.uniovi.notaneitor.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HomeController {

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

A continuación, 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.

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<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: #007bff;
  text-align: center;
  line-height: 60px
}
/* Sticky footer styles----- */
```

Ahora volvemos a desplegar y probamos de nuevo la nuestra web desde <http://localhost:8090/>

## 1.14 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 e 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





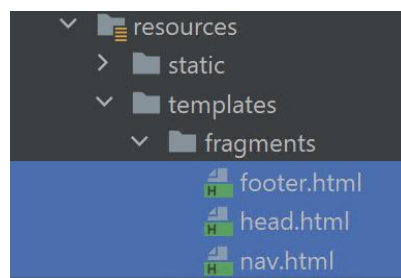
de código, por lo que modificar el diseño de un gran número de vistas en grandes aplicaciones puede volverse incómodo.

### ***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 código "común", **head.html**, **nav.html** y **footer.html**.



```
<!-- head.html -->
<head>
  <title>Notaneitor</title>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
  <!-- Bootstrap CSS -->
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
  <!-- Font Awesome CSS -->
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.0/css/all.css"
    integrity="sha384-lZN37f5QGtY3VHgisS14W3ExzMWZxybE1SJSesQp9S+oqd12jhcu+A56Ebc1zFSJ"
    crossorigin="anonymous">
  <!-- JS files: jQuery primero y luego Bootstrap JS -->
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
  <!-- Nuestra CSS -->
  <link rel="stylesheet" href="/css/custom.css" />
</head>
```

```
<!-- nav.html -->
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <a class="navbar-brand" href="#"></a>
  
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#myNavbar">
```





```
aria-controls="navbarColor02" aria-expanded="false" aria-label="Toggle navigation">
<span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="myNavbar">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item">
      <a class="nav-link" href="/home">Home<span class="sr-only">(current)</span></a>
    </li>
    <li class="nav-item dropdown active">
      <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-
toggle="dropdown"
      aria-haspopup="true" aria-expanded="false"> Gestión de notas
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="/mark/list">Ver Notas</a>
      <a class="dropdown-item" href="/mark/add">Agregar Nota</a>
      <div class="dropdown-divider"></div>
      <a class="dropdown-item" href="/mark/filter">Filtrar</a>
    </div>
    </li>
    <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle" href="#" id="profesorDropdown" role="button" data-
toggle="dropdown"
      aria-haspopup="true" aria-expanded="false"> Gestión de profesores
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="/professor/list">Ver profesores</a>
    </div>
    </li>
  </ul>
  <ul class="navbar-nav justify-content-end">
    <li class="nav-item">
      <a class="nav-link" href="/registrarse">
        <i class="fas fa-user-alt" style="font-size:16px"></i>
        Registrarse</a>
    </li>
    <li class="nav-item"><a class="nav-link" href="/identificarse">
      <i class="fas fa-sign-in-alt" style="font-size:16px"></i>
      Identifícate</a></li>
  </ul>
  <!--<form class="form-inline">
    <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-label="Search">
    <button class="btn btn-outline-light my-2 my-sm-0" type="submit">Search</button>
  </form> -->
</div>
</nav>
```

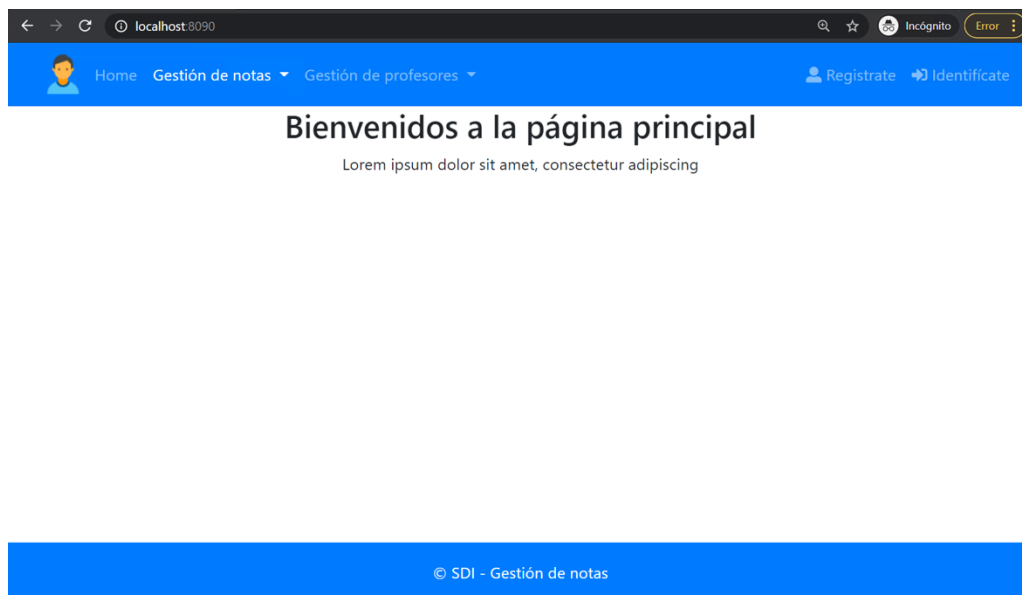
```
<!-- footer.html -->
<footer class="footer">
  <div class="container">
    <span class="text-white bg-primary">&copy; SDI - Gestión de notas</span>
  </div>
</footer>
```



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" xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://thymeleaf.org">
<head th:replace="fragments/head"></head>
<body>
  <nav th:replace="fragments/nav"></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"></footer>
</body>
</html>
```

Ahora volvemos a desplegar, accedemos a la raíz de nuestra página web <http://localhost:8090> y el resultado final de la página principal de la aplicación Web será como la siguiente imagen:



**Nota: Incluir el siguiente Commit Message ->**

**“SDI-IDGIT-2.6-Página de inicio y plantillas con Thymeleaf.”**

**OJO: sustituir IDGIT por tu número asignado (p.e. 2021-101):**

**“SDI-2021-101-2.6-Página de inicio y plantillas con Thymeleaf.”**

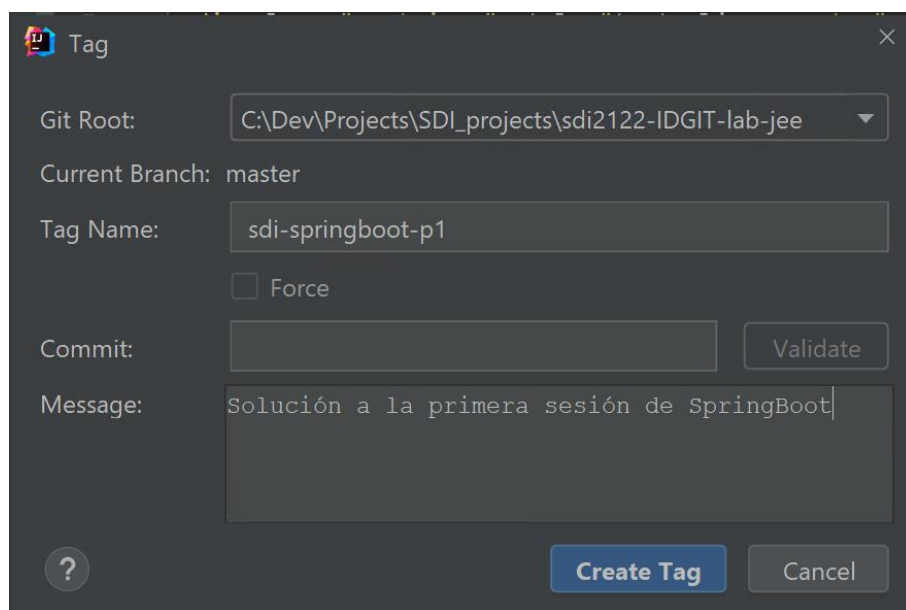
**(No olvides incluir los guiones y NO incluyas BLANCOS)**



## 1.13 Etiquetar proyecto en GitHub

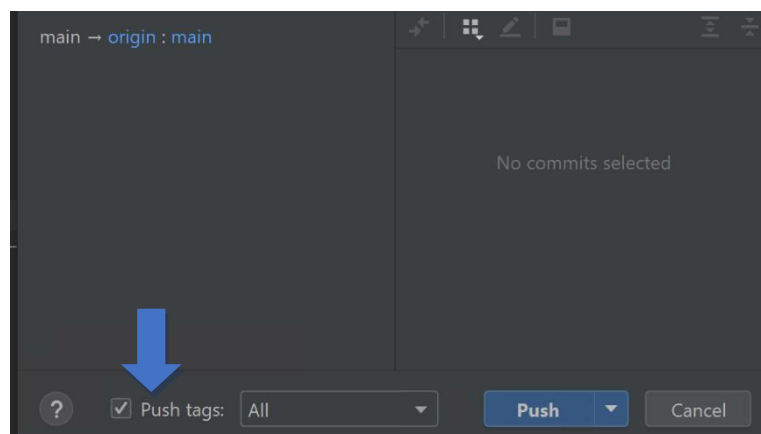
Para crear una **Release** en un proyecto que está almacenado en un repositorio de control de versiones lo común es utilizar etiquetas (Tags), que nos permitirá marcar el avance de un proyecto en un punto dado (una funcionalidad, una versión del proyecto, etc. Para crear una etiqueta en el proyecto en GitHub lo primero que vamos a hacer es hacer click derecho sobre el proyecto en IntelliJ IDEA e ir a la opción de **Git | New 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**.



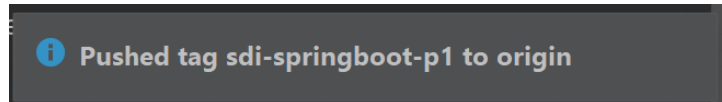
Con el paso anterior hemos creado la etiqueta en el repositorio local. Ahora tenemos que subir esa etiqueta al repositorio remoto (GitHub). Para subir una etiqueta al repositorio de Github hacemos click derecho sobre el proyecto e ir a la opción de **Git | Push**.

Una vez en la siguiente ventana seleccionamos la opción **Push tags** y pulsamos el botón **Push**.

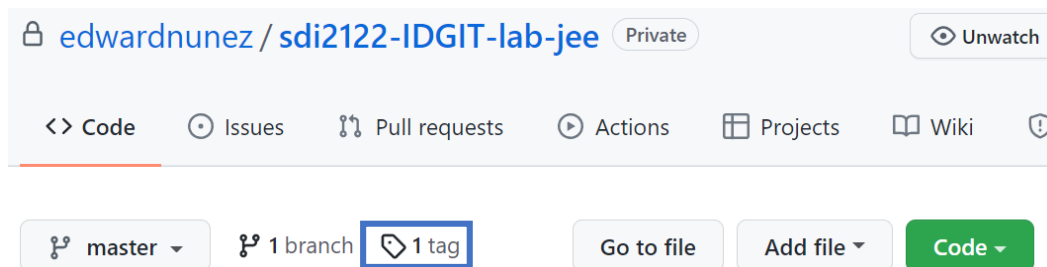




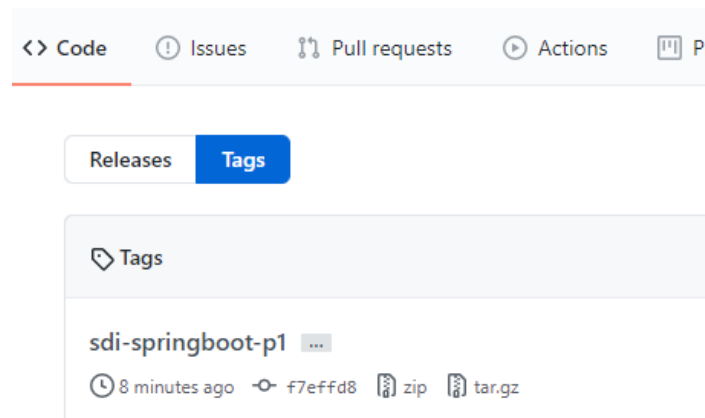
Al finalizar IntelliJ nos mostrará un mensaje en la parte inferior derecha con el resultado del push, tal como se muestra en la siguiente imagen:



Una vez subida las etiquetas al repositorio, podemos verificar dichas etiquetas en nuestra cuenta y repositorio de GitHub. **Vamos al repositorio correspondiente y pulsamos en el enlace de tags.**



Si se ha realizado la operación correctamente, debería ser visible la etiqueta creada anteriormente:



## 1.15 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.

- ⇒ SDI-2021-101-2.1-Create Spring project.
- ⇒ SDI-2021-101-2.2-Trabajando con Controladores.
- ⇒ SDI-2021-101-2.3-Beans, servicios e inyección de dependencias.
- ⇒ SDI-2021-101-2.4-Modelo, acceso a datos simple.
- ⇒ SDI-2021-101-2.5-Vistas, motor de plantillas Thymeleaf.
- ⇒ SDI-2021-101-2.6-Página de inicio y plantillas con Thymeleaf.