

Tabla de contenidos

Spring Boot	2
Introducción	2
Contenedor de aplicaciones integrado	3
Starters	3
Preparar el entorno de trabajo	4
Tu primera aplicación con Spring Boot	5
Cómo crear una aplicación con SpringBoot Initializr	5
Recursos generados por SpringBoot Initializr	11
Lanzar el proyecto	12
Introducción a los controladores con Spring Boot	13
Entendiendo el funcionamiento de un controller	13
Creando la estructura de paquetes (packages)	14
Creando mi primer controlador	15
Diferencia entre @RestController y @Controller	17
Devolver una página HTML desde el controlador	17
Uso de @RequestMapping	18
Mapeo de más de una URI a un controlador	19
Redirigir a otro controlador	20
Pasar parámetros a nuestra plantilla	20
Thymeleaf	24
Expresiones básicas	24
Expresiones variables	24
Expresiones de selección	25
Operadores aritméticos	26
Operadores textuales	26
Operadores relacionales	26
Operadores booleanos	26
Comparaciones	26
Operadores condicionales	26
Expresiones por defecto	27
Elemento th:block (o th-block)	27
Iterar con Thymeleaf	27
Estado de la iteración	28
Rescatar parámetros de la ruta	28
@RequestParam	28
@PathVariable	31

Spring Boot

Introducción

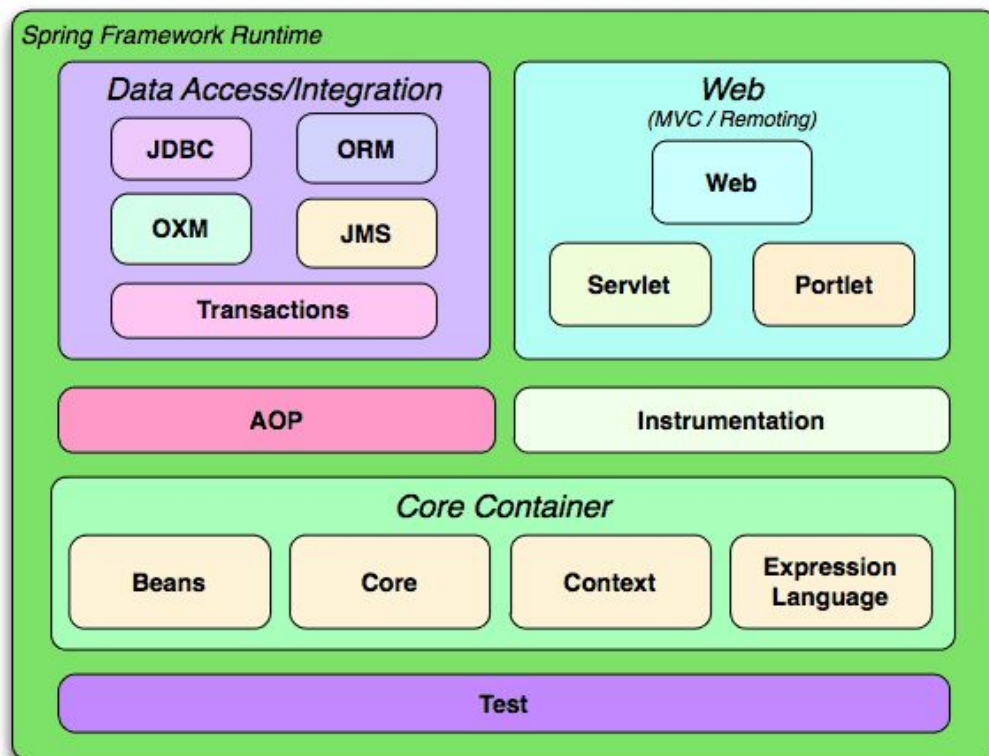
Si has desarrollado con Java en algún momento en los últimos años, seguramente te suene Spring Framework, aunque sea de oídas.

Spring Framework es un framework Open Source que facilita la creación de aplicaciones de todo tipo en Java, Kotlin y Groovy.

Spring Framework está dividido en diversos módulos que podemos utilizar, ofreciéndonos muchas más funcionalidades:

- Core container: proporciona inyección de dependencias e inversión de control.
- Web: nos permite crear controladores Web, tanto de vistas MVC como aplicaciones REST.
- Acceso a datos: abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (Object XML Mappers), JMS y transacciones.
- Programación orientada a Aspectos (AOP): ofrece el soporte para aspectos.
- Instrumentación: proporciona soporte para la instrumentación de clases.
- Pruebas de código: contiene un framework de testing, con soporte para JUnit y TestNG y todo lo necesario para probar los mecanismos de Spring.

Estos módulos son opcionales, por lo que podemos utilizar los que necesitemos sin tener que llenar nuestro **classpath** con clases que no vamos a usar.



Esquema que ilustra los diferentes módulos de Spring, obtenido de la documentación oficial

Hay cientos de tecnologías que Spring permite integrar. Desde bibliotecas que implementan opentracing hasta las que nos generan métricas para nuestra aplicación, pasando por serialización/deserialización a JSON y XML, seguridad con OAuth2 o programación reactiva entre otras.

En general, Spring aumenta la productividad y reduce la fricción al ofrecernos abstracciones sobre implementaciones de tecnologías concretas. Un ejemplo claro es el de spring-data, que nos permite definir el acceso a base de datos con interfaces Java. Esto lo consigue parseando el nombre de los métodos y generando la consulta con la sintaxis específica para el driver que utilicemos. Por ejemplo, cambiar nuestra aplicación de MySQL a PostgreSQL es tan sencillo como cambiar el driver: Spring se encarga de la sintaxis de forma transparente.

A pesar de "la magia de Spring", como muchos lo llaman, Spring nos permite desactivar estos "comportamientos mágicos" en caso de ser necesario, por lo que podemos tomar el control cuando necesitemos más granularidad. Siguiendo con el ejemplo de spring-data, este control sería necesario si tenemos que realizar consultas mucho más complejas que un `SELECT * BY name`. En esos casos, entre otras opciones, podemos anotar nuestro método con `@Query` y escribir la consulta que deseemos.

Por lo general, Spring no obliga a implementar ni extender nada, lo que nos permite escribir código que es "agnóstico" del framework. De esta forma, los desarrolladores con cero o muy poco conocimiento de Spring pueden realizar su trabajo sin mayores complicaciones.

Spring es de código abierto y tiene una gran comunidad detrás. Si encuentras un bug, echas en falta una funcionalidad o lo que sea, siempre puedes abrir un ticket o contribuir por tu cuenta.

Si bien es cierto que Spring Framework es muy potente, **la configuración inicial y la preparación de las aplicaciones para producción** son tareas bastante tediosas. **Spring Boot simplifica el proceso al máximo** gracias a sus dos principales mecanismos.

Contenedor de aplicaciones integrado

Spring Boot permite compilar nuestras aplicaciones Web como un archivo **.jar** que podemos ejecutar como una aplicación Java normal (como alternativa a un archivo **.war**, que desplegaríamos en un servidor de aplicaciones como Tomcat).

Esto lo consigue **integrando el servidor de aplicaciones** en el propio **.jar** y levantándolo cuando arrancamos la aplicación. De esta forma podemos distribuir nuestras aplicaciones de una forma mucho más sencilla al poder configurar el servidor junto con la aplicación.

Nota: Spring boot permite distribuir tu aplicación como un jar, no lo impone. Si prefieres desplegar tu aplicación en un servidor de aplicaciones tradicional, Spring Boot te deja compilar el código como un **.war** que no incluya ningún servidor de aplicaciones integrado.

Starters

Spring Boot nos proporciona una serie de dependencias, llamadas starters, que podemos añadir a nuestro proyecto dependiendo de lo que necesitemos: crear un controlador REST,

acceder a una base de datos usando JDBC, conectar con una cola de mensajes Apache ActiveMQ, etc.

Una vez añadimos un starter, éste nos proporciona todas las dependencias que necesitamos, tanto de Spring como de terceros. Además, los starters vienen configurados con valores por defecto, que pretenden minimizar la necesidad de configuración a la hora de desarrollar.

Al igual que con Spring Framework, **cualquier configuración puede ser modificada** de ser necesario.

Preparar el entorno de trabajo

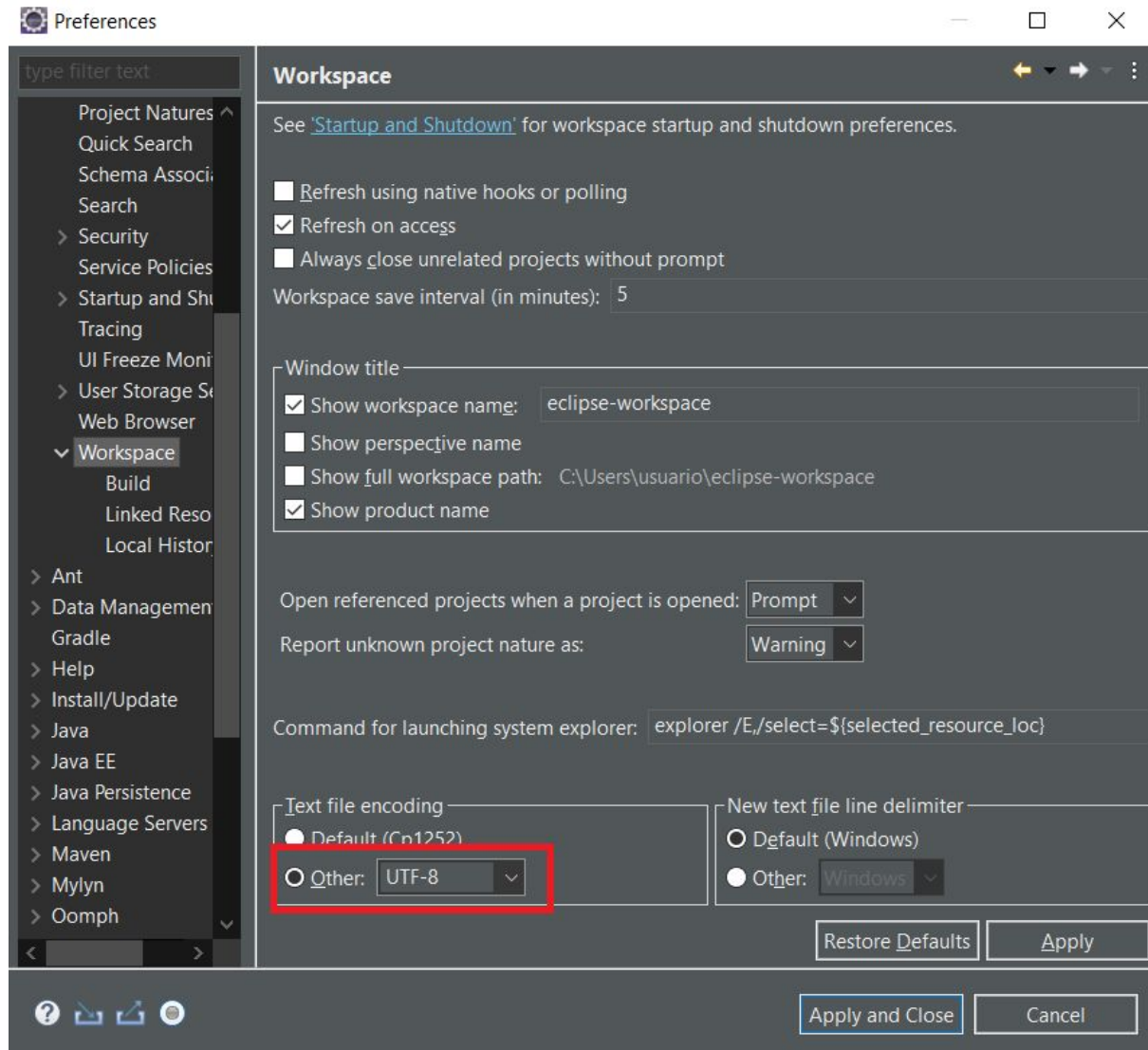
Para trabajar con Spring Boot vamos a usar como IDE de desarrollo Eclipse. Este IDE es gratuito y nos lo podemos descargar de la siguiente url <https://www.eclipse.org/downloads/>

De entre las versiones disponibles se aconseja bajar la que ya incluye el JRE. Esta versión hará que no tengamos que instalar nada más ya que viene con la máquina virtual de JAVA, imprescindible para hacer funcionar nuestros proyectos.

La instalación es sencilla y puede tardar más o menos tiempo dependiendo de la conexión a Internet que tengamos. Tras ejecutar el instalador de eclipse lo primero que nos pedirá es la versión a instalar. Hay que seleccionar la opción Eclipse IDE for Enterprise Java Developers



Una vez instalado Eclipse se aconseja cambiar la codificación de los ficheros a UTF8. Para ello hay que ir a Preferences -> General ->Workspace -> Text File Encoding y en la opción Other cambiar el valor a UTF8.



Tu primera aplicación con Spring Boot

Vamos a ver paso a paso cómo crear un proyecto en blanco, como plantilla para comenzar cualquier desarrollo web. Utilizamos Java y Spring Boot para construir una aplicación web desde cero con sencillos pasos usando SpringBoot Initializr.

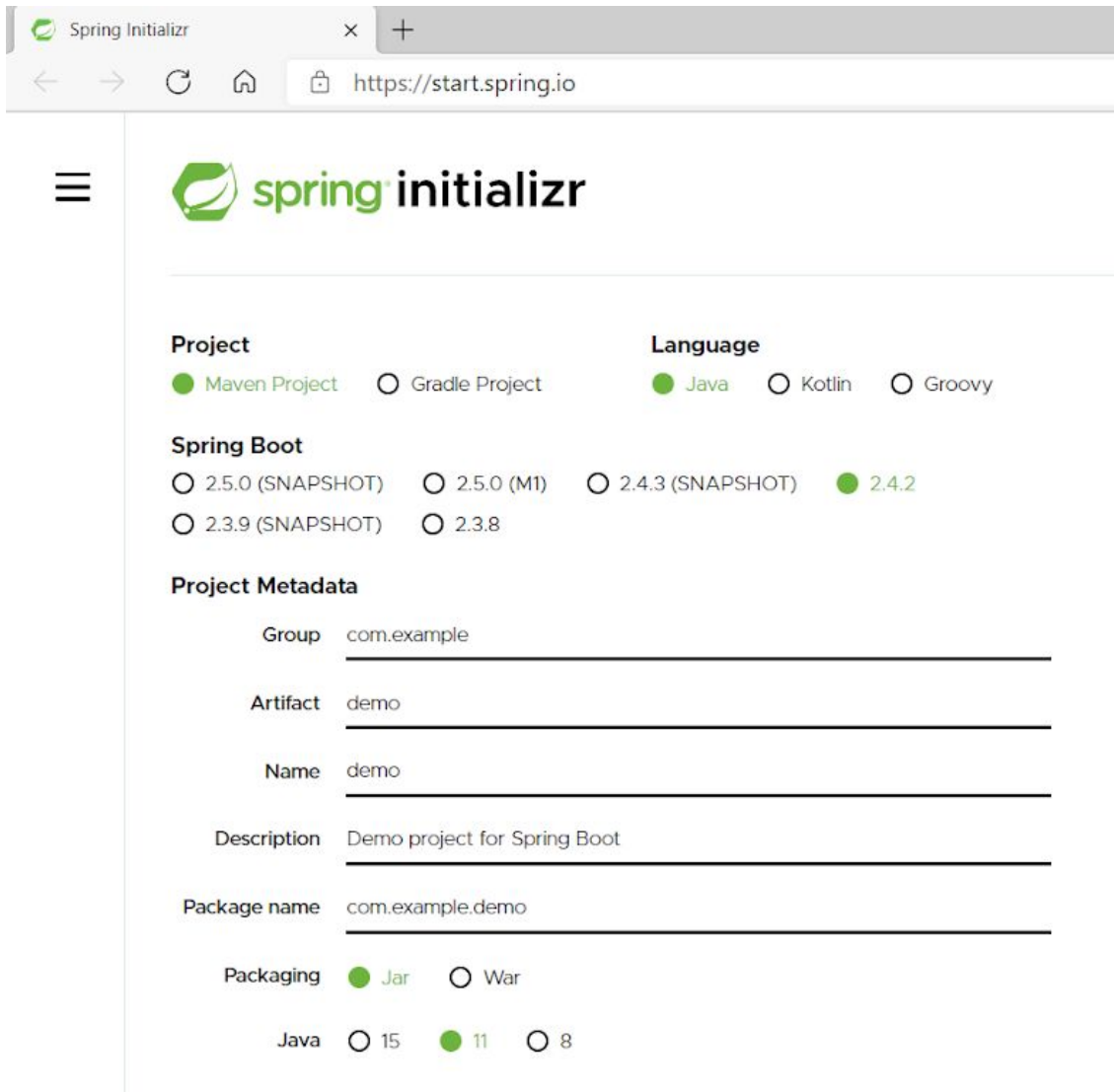
Cómo crear una aplicación con SpringBoot Initializr

Por parte de los desarrolladores de Spring, se pone a disposición de los programadores una herramienta web online denominada Spring Initializr donde por medio de unos parámetros de configuración, se genera automáticamente un proyecto Maven o Gradle, según se elija, en un archivo comprimido ZIP. Este ZIP contiene la estructura de la aplicación para ser importada directamente a un editor de programación como Eclipse IDE, Netbeans IDE o IntelliJ.

NOTA: Nosotros usaremos la aplicación web, aunque se puede instalar Spring Tools para Eclipse utilizando su Marketplace. Para ello desde el Marketplace de Eclipse basta con buscar “Spring Tools” e instalar la entrada “Spring Tools 4”.

A la aplicación web se accede mediante la siguiente dirección: <https://start.spring.io/>.

El asistente web solicita una serie de datos necesarios para poder ejecutar la plantilla que construye los primeros archivos del programa. Para todos ellos aporta una configuración por defecto que conviene cambiar, como el nombre de la aplicación, la versión de java a utilizar o el package que se usará en las clases generadas.



The screenshot shows the Spring Initializr web application in a browser. The browser's address bar displays <https://start.spring.io/>. The page features the Spring Initializr logo and a sidebar menu. The main content area is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions **2.5.0 (SNAPSHOT)**, **2.5.0 (M1)**, **2.4.3 (SNAPSHOT)**, **2.4.2** (selected), and **2.3.9 (SNAPSHOT)**, **2.3.8**.
- Project Metadata:** A form with the following fields:
 - Group:** com.example
 - Artifact:** demo
 - Name:** demo
 - Description:** Demo project for Spring Boot
 - Package name:** com.example.demo
 - Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
 - Java:** Includes radio buttons for versions **15**, **11** (selected), and **8**.

A continuación se explica qué parámetros hay y para qué sirven:

- **Project:** Permite elegir la herramienta de construcción de la aplicación. En Java las dos herramientas más usadas son Maven y Gradle. Recomendamos Maven al ser la más expandida.

- **Language:** Lenguaje de programación que se va a utilizar en la aplicación. Los tres tipos están soportados por la máquina virtual JVM. Java es la opción más extendida y tiene mejor soporte de los editores de programación.
- **Spring Boot:** Versión del Spring Boot a usar. Siempre que se pueda se optará por la última estable, compuesta únicamente por números.
- **Project Metadata, Group:** Se refiere al descriptor de Maven groupId, utilizado para clasificar el proyecto en los repositorios de binarios. Normalmente se suele usar una referencia similar a la de los packages de las clases.
- **Project Metadata, Artifact:** Se refiere al otro descriptor de Maven artifactId, y por tanto para indicar el nombre del proyecto y del binario resultante. La combinación de groupId y artifactId (más la versión) identifican inequívocamente a un binario dentro de cualquier organización.
- **Packaging:** Indica qué tipo de binario se debe construir. Si la aplicación se ejecutará por sí sola se seleccionará JAR. Este JAR contiene todas las dependencias dentro de él y se podrá ejecutar con **java -jar binario-<version>.jar**. Si por el contrario, la aplicación se ejecutará en un servidor J2EE existente o en un Tomcat ya desplegado se deberá escoger WAR.
- **Java:** Se selecciona la versión de Java a usar. En este caso, hay que tener cuidado con la versión de Java a usar. Si se selecciona la más reciente puede ocurrir que no se garantice la compatibilidad con otras librerías o proyectos que se quieran incluir.

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf

TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

- **Dependencies:** Buscador de dependencias con los starters de Spring boot disponibles. Las dependencias más habituales son:
 - Spring Web se deberá escoger cuando se desee hacer una aplicación web o microservicios, siempre que se requiera una comunicación http y por tanto el uso de Spring MVC.
 - Thymeleaf Incorpora el motor de plantillas para HTML dinámico, sucesor de los anteriores JSP (Java Server Page).
 - Spring Data JPA necesario para utilizar la capa estándar de acceso a base de datos SQL denominada Java Persistence Api.
 - Spring Security Permite incorporar controles de acceso en base a usuarios y roles sobre URLs de la aplicación. También habilita el control de ejecución de métodos de servicio en base a roles según los estándares J2EE.

- Mysql/Postgresql Incluye el JAR que contiene el driver JDBC necesario para configurar la capa de JPA según la base de datos a usar.
- Otras... El asistente permite seleccionar entre más de 50 dependencias e integraciones de herramientas open source dentro de los proyectos realizados con Spring.

Una vez seleccionados los parámetros que se quieren, haciendo click en el botón «Generate-Ctrl+» se descarga un archivo ZIP con el nombre del Artifact que contendrá la carpeta con la estructura de la aplicación lista para importar desde el IDE.

Nosotros crearemos un proyecto de Spring Boot con las siguientes características:



Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.5.0 (SNAPSHOT) ☐ 2.5.0 (M1) ☐ 2.4.3 (SNAPSHOT) ☒ 2.4.2
☐ 2.3.9 (SNAPSHOT) ☐ 2.3.8

Project Metadata

Group dwes.java.spring

Artifact proyecto

Name proyecto

Description DWES Proyectos en Spring Boot

Package name dwes.java.spring.proyecto

Packaging ☒ Jar ☐ War

Java ☒ 15 ☐ 11 ☐ 8

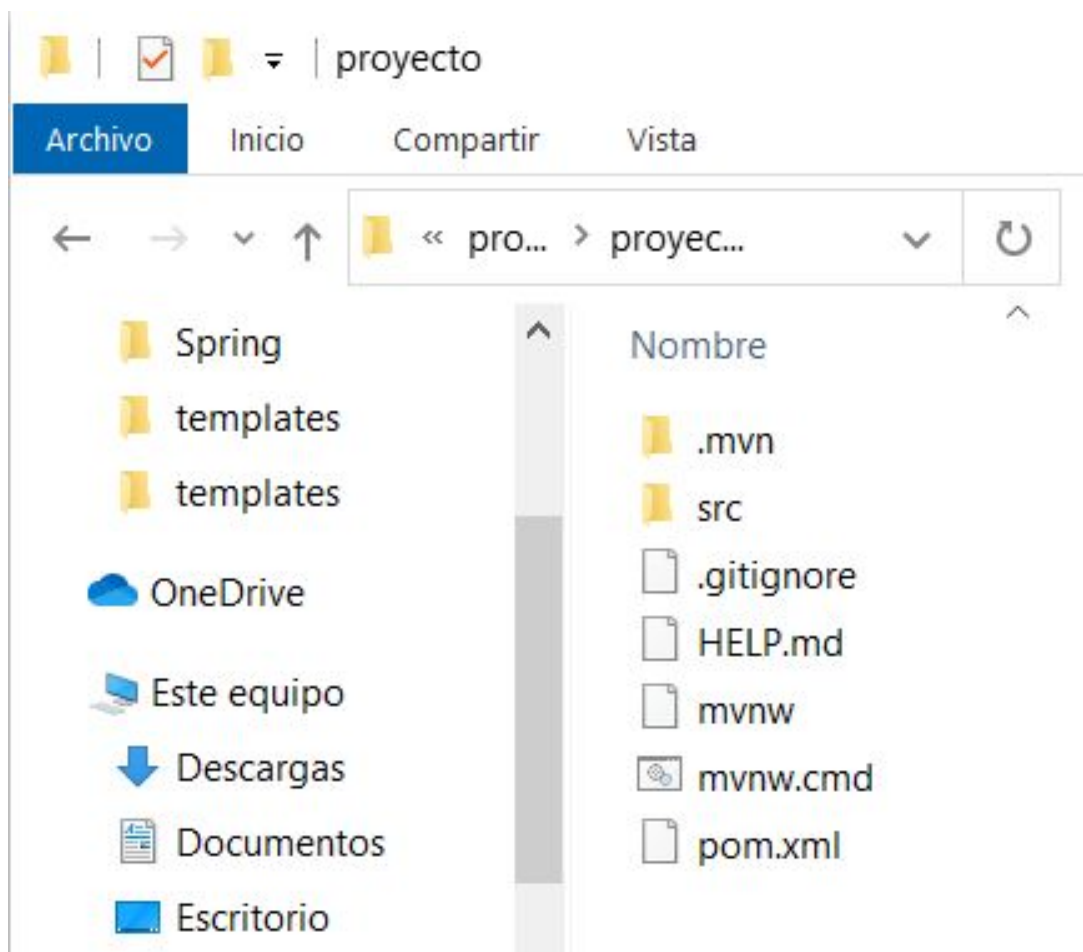
Dependencies**ADD DEPENDENCIES...** CTRL + B**Spring Web****WEB**

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf**TEMPLATE ENGINES**

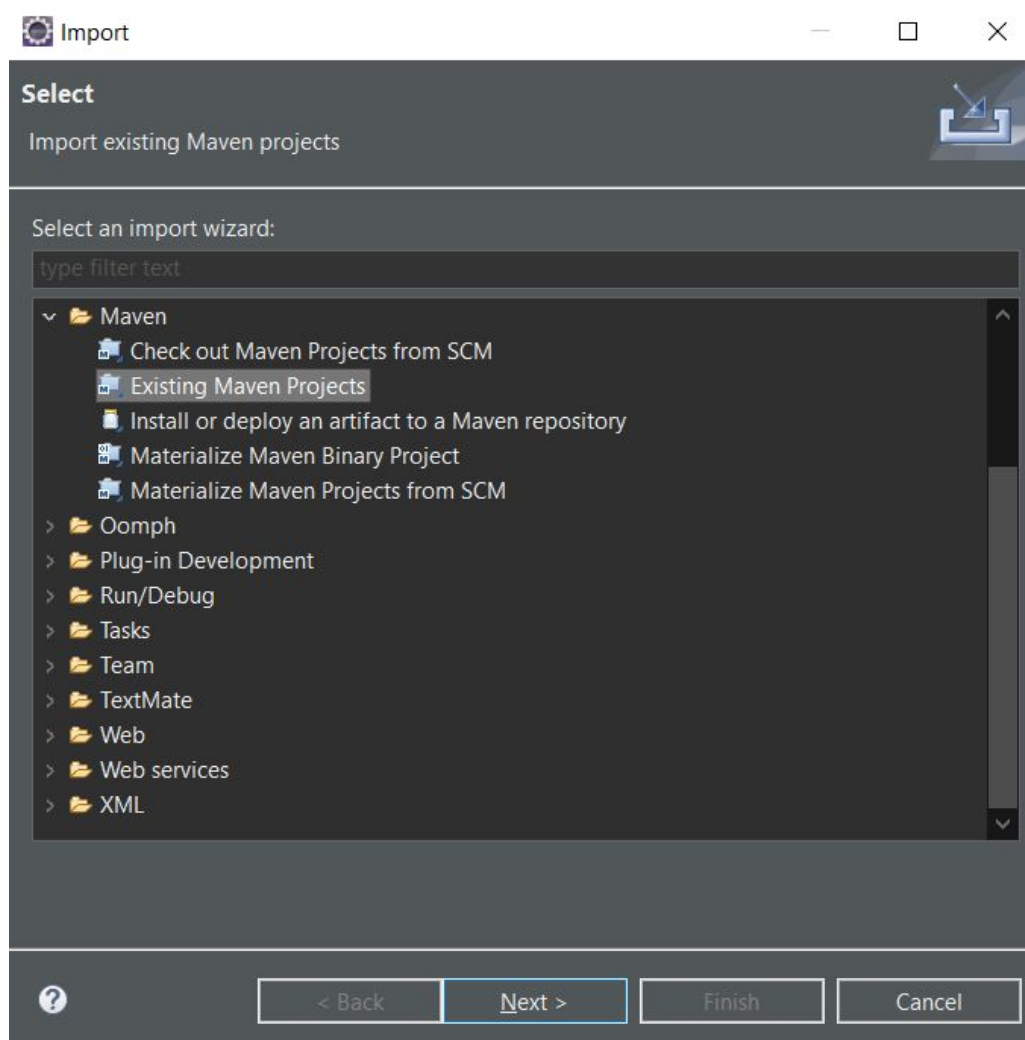
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Una vez descargado el ZIP si lo descomprimos tendrá la siguiente estructura

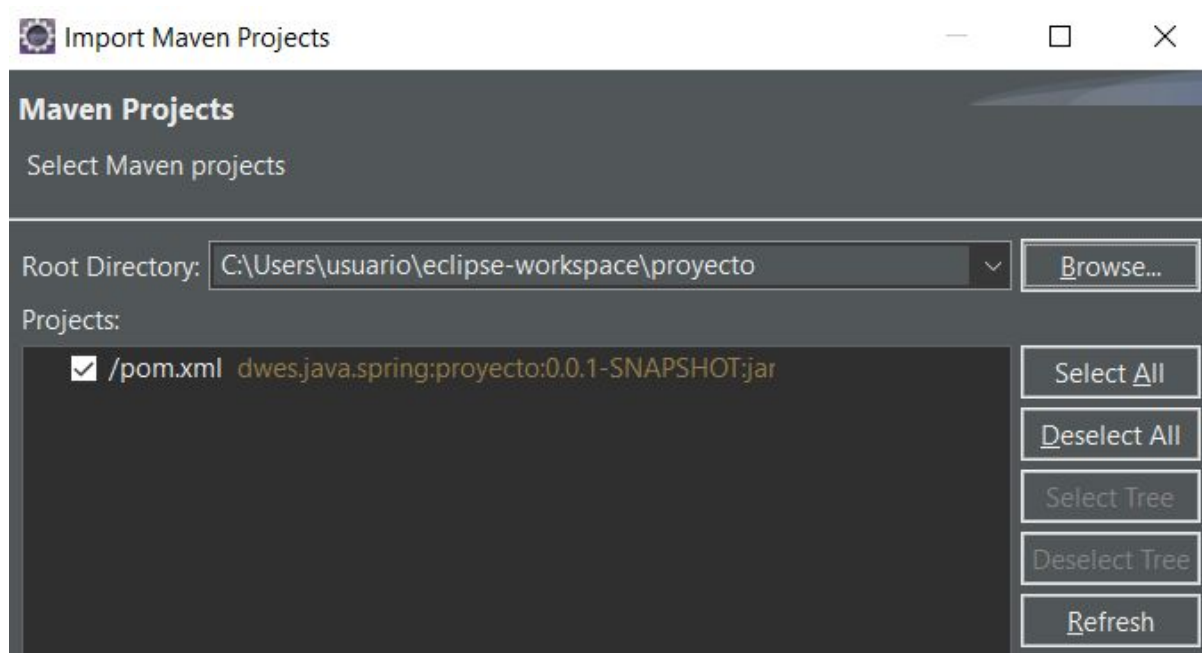


Para tener nuestro proyecto en Eclipse bastará con importar un proyecto Maven existente.

NOTA: Si queremos que nuestro proyecto esté en nuestro workspace se tendrá que copiar en el mismo antes de realizar la importación ya que en caso contrario nuestro proyecto estará ubicado desde lo hayamos importado.



Tras seleccionar la carpeta de nuestro proyecto Maven, proyecto en nuestro caso, tendremos que seleccionar el fichero pom.xml y finalizar.



Recursos generados por SpringBoot Initializr

Tras la importación el editor creará y configurará el classpath y las librerías que se indican en él para que estén disponibles para el programador.

El contenido del proyecto en Eclipse tendrá la siguiente estructura:

```
.
├── HELP.md
├── mvnw
├── mvnw.cmd
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── dwes
│   │   │   │   ├── java
│   │   │   │   │   ├── spring
│   │   │   │   │   │   └── SpringAppApplication.java
│   │   │   └── resources
│   │   │       ├── application.properties
│   │   │       ├── static
│   │   │       └── templates
│   └── test
│       ├── java
│       │   ├── dwes
│       │   │   ├── java
│       │   │   │   ├── spring
│       │   │   │   │   └── SpringAppApplicationTests.java
```

De los ficheros incluidos, el más importante es **SpringAppApplication.java** que corresponde con el punto de entrada a la ejecución del programa. Aquí es donde se aloja el

método `public static void main(String[] arg)` que es el que inicializa toda la aplicación de Spring Boot.

Durante el arranque del framework, Spring revisa el resto de directorios o paquetes que cuelgan de método `main` en búsqueda de clases marcadas con alguna anotación que permite el registro de componentes como: `@Service`, `@Component`, `@Repository` y demás anotaciones de Spring y de Spring MVC como `@Controller` o `@RestController`. El paquete utilizado es el **com.java.spring**, opción que se especifica en el asistente Spring Initializr.

El siguiente fichero más importante es **application.properties** que es donde se aloja toda la configuración de los componentes de Spring Boot, como qué encoding usar (utf-8), si debe usar caché en las plantillas, puerto usado, etc... En el enlace de [configuraciones y propiedades comunes de spring boot](#) hay una referencia de todas las existentes y sus valores iniciales.

El siguiente fichero es **SpringAppApplicationTests.java** donde se aloja el primer test de ejemplo que genera Initializr por nosotros. Ahí, el usuario debe añadir los test que considere oportunos.

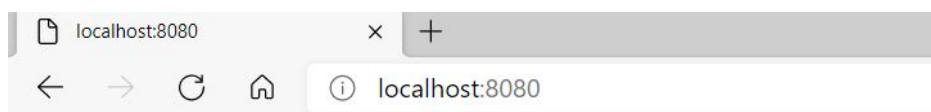
Los otros directorios son **static** que es donde se alojan los recursos estáticos que la aplicación debe servir sin procesar. Por ejemplo en esta carpeta se incluyen los ficheros CSS, Javascript, imágenes o fuentes que se referencian desde el HTML.

Si el fichero alojado en este directorio está en una subcarpeta de `static`, como puede ser `css`, de forma que la ruta sea `static/css/main.css`. Spring Boot lo publicará automáticamente en la url `http://localhost:8080/css/main.css`. Así que no debe haber información sensible que cuelgue del directorio `static` porque entonces será accesible a todo el mundo.

Y por último la carpeta **templates** almacena los ficheros que permiten generar HTML dinámicamente con algún motor de plantillas soportado por Spring como Thymeleaf o Freemarker.

Lanzar el proyecto

El proyecto creado es muy simple, si tratamos de arrancar el método `main` de la clase **SpringAppApplication**, veremos cómo Spring inicializa toda la aplicación y se pone a la escucha del puerto 8080, pero si dirigimos el navegador a esa URL no aparecerá ningún contenido. En su lugar aparecerá:



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Jan 31 15:16:27 CET 2021

There was an unexpected error (type=Not Found, status=404).

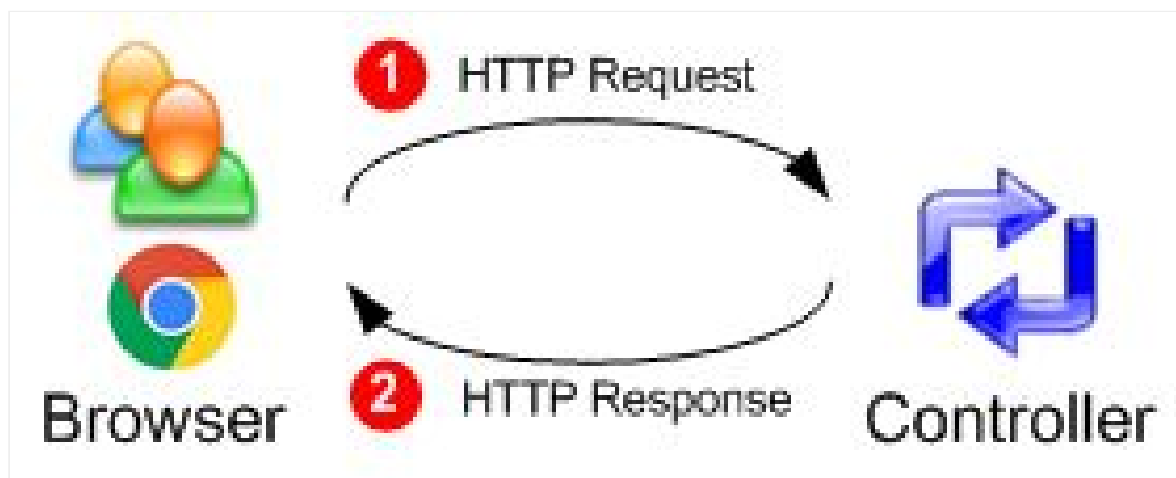
Introducción a los controladores con Spring Boot

Un controlador, aunque es más conocido con el término controller (en inglés), es el encargado de responder a las peticiones. Normalmente, estas peticiones son realizadas por el usuario, aunque también pueden ser peticiones más automatizadas como APIs, páginas front con Ajax, etc.

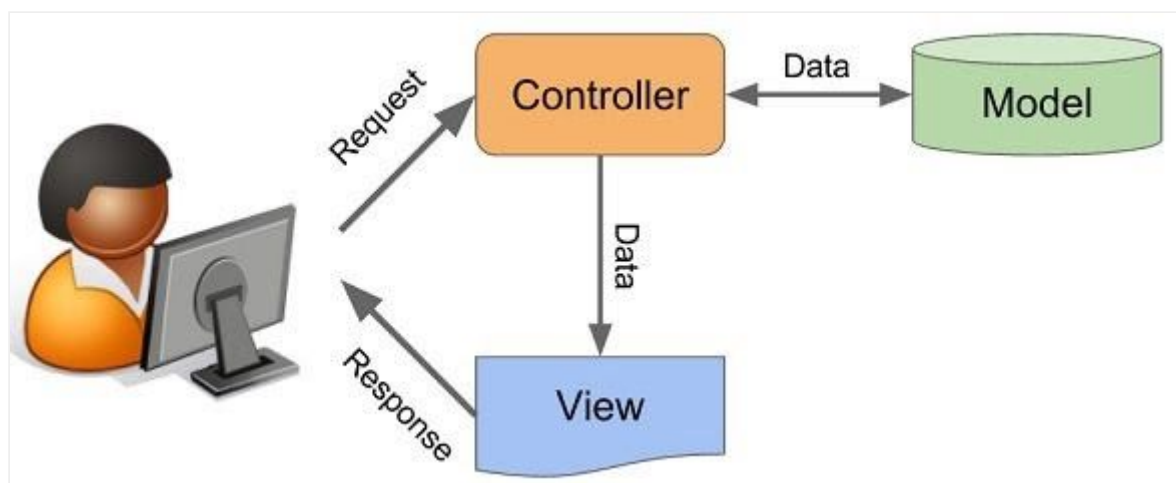
El controlador, en la vida real podría ser un recepcionista de un hotel que nos manda a un punto u otro, dependiendo de la petición (request) que le enviemos. El controlador por tanto nos permite orquestar, es el encargado de recibir una petición que se envía a nuestro servidor. Lo que estábamos haciendo hasta ahora, era lanzar una petición que no llegaba a ningún punto. Esto se produce, debido a que no teníamos un controlador que pudiera capturar dicha petición. Por tanto, el request no tenía respuesta (response) y nos aparecía un error de página en blanco.

Entendiendo el funcionamiento de un controller

Si creamos un controlador, ya podremos responder a dicha ruta y por tanto, ya podremos mostrar un mensaje.



El objetivo ahora mismo es familiarizarnos y comunicarnos únicamente con el controlador, aunque lo habitual es trabajar con el MVC.



Creando la estructura de paquetes (packages)

Vamos a crear un controlador, pero antes, tenemos que crear un **package**, que contenga dicho controlador.

Contenido del fichero SpringAppApplication.java

```
package dwes.java.spring.proyecto;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

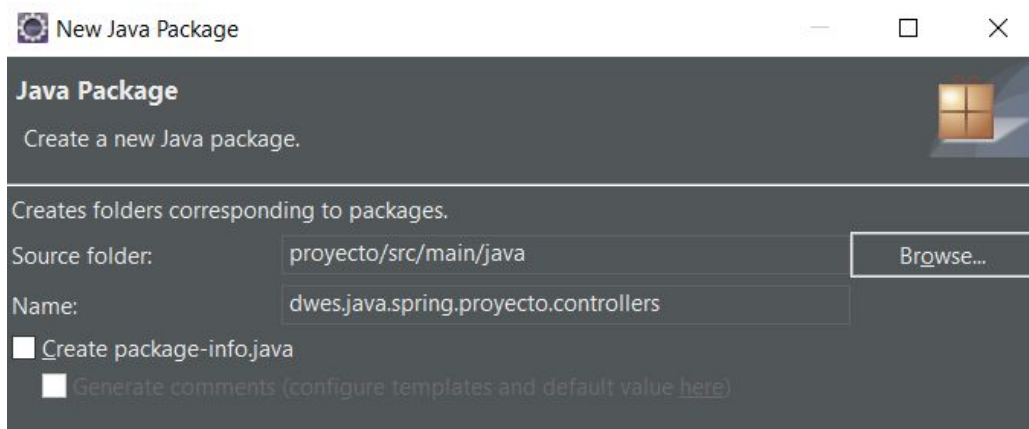
@SpringBootApplication
public class ProyectoApplication {

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

@SpringBootApplication es una anotación que ejecuta tres anotaciones de autoconfiguración con sus valores por defecto:

- @EnableAutoConfiguration
- @ComponentScan
- @Configuration

Si no queremos tener que indicarle que escanee varios packages (lo cual suele ser la mejor idea), solemos meter toda la estructura de paquetes dentro del package “padre” que será el package donde tenemos el main. Para crear por tanto el nuevo paquete, pulsamos botón derecho sobre el package que contiene el main. De esta forma, cuando vayamos a poner el nombre directamente nos saldrá el **dwes.java.spring.proyecto** y solo le tendremos que añadir un **.** y el nombre del package hijo, en este caso **controllers**. Sino, tendríamos que escribir el nombre completo del package.



Los controladores, los separaremos en un package llamado controllers. Ya que tenemos varios packages, vamos a organizarlos de manera que el propio nombre sea tan intuitivo, que nos indique lo que se almacena en su interior.

Con esto ya tenemos el package creado. El paquete estará vacío y por tanto aparecerá con el icono de la caja en blanco, en lugar de marrón.

Creando mi primer controlador

Si creamos una clase dentro del package, el icono se pondrá en marrón. Es la manera que tiene el IDE de diferenciar los package vacíos de los que contienen clases. Le ponemos el nombre al controlador. En este caso, al encargarse de las rutas del index usaremos un nombre que lo identifique como Index y como Controlador. Por tanto, el nombre será **IndexController**.

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

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

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

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

☐ Generate comments

A continuación vamos a añadir código a nuestro controlador para muestre un mensaje por pantalla en lugar del error.

El código de nuestro IndexController será el siguiente:

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class IndexController {
    @GetMapping("/")
    @ResponseBody()
    public String index() {
        return "Página de Inicio";
    }
}
```

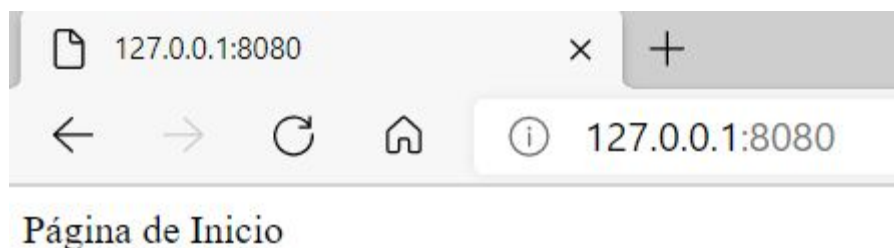
Si analizamos lo anterior, tenemos:

`package dwes.java.spring.proyecto.controllers`: define el paquete donde está albergada la clase.

`@Controller`: define que la clase es un controlador. Es una especialización de `@Component` y se detecta automáticamente a través de la exploración de classpath.

`@GetMapping("/")`: define la ruta que mapeará dicho elemento del controlador. En este caso, será escuchará a una petición de tipo GET, que apunta a <http://localhost:8080/>

`@ResponseBody`: la anotación `@ResponseBody` le dice a un controlador que el objeto devuelto se serialice automáticamente en JSON y se devuelve al objeto `HttpResponse`.



El código anterior es equivalente a

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
```

```
@RestController
public class IndexController {

    @GetMapping("/")
    public String index() {
        return "Página de Inicio";
    }
}
```

Diferencia entre @RestController y @Controller

Ahora que estás familiarizado con estas dos anotaciones, es un buen momento para analizar algunas diferencias de entre @RestController y @Controller.

El @Controller es una anotación común que se utiliza para marcar una clase como Controlador MVC de Spring mientras que @RestController es un controlador especial utilizado en los servicios web RESTful y el equivalente a @Controller + @ResponseBody.

Una de las diferencias clave entre @Controller y @RestController en Spring MVC es que una vez que marcas una clase @RestController entonces cada método devuelve un objeto en lugar de una vista.

Devolver una página HTML desde el controlador

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class IndexController {
    @GetMapping("/")
    public String index() {
        return "index";
    }
}
```

Nada más ejecutarse el método del controlador, Spring pasará el control a la vista con nombre index correspondiente al fichero index.html de la carpeta de templates, tal y como indica el return «index», en el que no hace falta indicar la extensión .html.

Para completar el círculo y ver los resultado necesitamos crear la vista index.html con el siguiente contenido:

Contenido de la página HTML

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
    <title>Mi primera aplicación</title>
  </head>
  <body>
    <h1>Hola desde mi primera página</h1>
  </body>
</html>
```

Uso de @RequestMapping

Esta es la anotación original para mapear cualquier tipo de verbo HTTP con un método.

De hecho, podríamos sustituir el código del controlador anterior por:

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class IndexController {
    @RequestMapping(value="/", method=RequestMethod.GET)
    public String index() {
        return "index";
    }
}
```

Podemos utilizar también la anotación @RequestMapping para definir un segmento de ruta a nivel de controlador. De esta forma todas las rutas están bajo este segmento.

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
```

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

@Controller
@RequestMapping("/app")
public class IndexController {

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

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

Las rutas anteriores serían:

- <http://127.0.0.1:8080/app/index>
- <http://127.0.0.1:8080/app/hola>

Mapeo de más de una URI a un controlador

La anotación `@RequestMapping` y sus derivadas (`@GetMapping`, `@PostMapping`, ...) pueden recibir más de una ruta como argumento. Lo hacen recibiendo varias entre `{ }`.

```
@GetMapping({"/", "/index", "/list"})
```

De esta forma, tanto si invocamos a `/`, como a `/index` o `/list`, todas las llamadas se harán al mismo método.

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class IndexController {

    @GetMapping({"/", "/index", "/list"})
    public String index() {
        return "index";
    }
}
```

Redirigir a otro controlador

Como hemos visto, la forma más sencilla de que un controlador nos lleve a una vista es devolviendo el nombre de la plantilla a renderizar como un String:

Sin embargo, habrá ocasiones en las que nos interese que un controlador nos lleve directamente a otro. Un escenario típico es, tras haber procesado un formulario (por ejemplo, de inserción de un nuevo registro); posiblemente, después de procesar esa petición, queramos visualizar el listado completo de registros, y así comprobar que el nuevo registro ha sido insertado.

Para poder hacer una redirección, incluimos la palabra `redirect`: en el valor de retorno del método, seguido de la ruta del controlador al cual queremos redirigirnos.

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class IndexController {
    @GetMapping("/")
    public String index() {
        return "index";
    }

    @GetMapping("/redirect")
    public String redirect() {
        return "redirect:/";
    }
}
```

Pasar parámetros a nuestra plantilla

A continuación vamos a ver como pasar parámetros desde nuestro controlador a nuestras plantillas HTML. Cómo se muestran los datos en nuestras plantillas HTML dependerá del motor de plantillas utilizado. En nuestro caso el motor de plantillas que vamos a utilizar es Thymeleaf, uno de los más extendidos.

Thymeleaf es, estrictamente hablando, un motor de plantillas, que se suele utilizar sobre todo en el contexto de HTML. Lo que hace es coger un documento HTML, que tiene dentro algunas expresiones, le aplica un procesamiento y produce contenido, que en este caso será contenido web.

No añade prácticamente ninguna etiqueta nueva (si no queremos, ninguna), sino lo que hace es añadir nuevos atributos a etiquetas ya conocidas.

Thymeleaf es casi tan legible como si fuera un documento HTML.

Controlador que pasa un parámetro a nuestra plantilla

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class IndexController {
    @GetMapping("/mensaje")
    public String mensaje(Model model) {
        model.addAttribute("nombre", "Antonio");
        return "mensaje";
    }
}
```

En este caso estamos pasando a nuestra página mensaje.html una variable nombre con valor “Antonio”

Nuestro fichero mensaje.html

```
<!DOCTYPE html>
<html lang="es" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
    <title>Mi primera aplicación</title>
</head>
<body>
    <h1>Hola <span th:text="{nombre}">nombre</span></h1>
</body>
</html>
```

Para que un HTML sea considerado como una plantilla de Thymeleaf la etiqueta html tiene que ser de la forma

```
<html lang="es" xmlns:th="http://www.thymeleaf.org">
```

La sintaxis de Thymeleaf es la siguiente

```
<h1>Hola <span th:text="${nombre}">nombre</span></h1>
```

A nuestra etiqueta, en este caso span, le añadimos el atributo th:text que se iguala con la variable que se recibe `${variable}` y que sustituye al contenido que hubiera en ella. Así el texto nombre del span se reemplazará por “Antonio”.

Vamos a ver un ejemplo en el que a nuestra vista vamos a pasarle un array.

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class IndexController {
    @GetMapping("/array")
    public String continente(Model model) {
        String[] continentesMundo = {
            "Africa", "Antártida", "Asia", "Australia",
            "Europa", "Ámerica del Norte", "Ámerica del Sur"
        };
        model.addAttribute("continentes", continentesMundo);
        return "continentes";
    }
}
```

En nuestra vista podemos recorrer la lista de elementos con la estructura **th:each**

```
<!DOCTYPE html>
<html lang="es" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8" />
        <title>Mi primera aplicación</title>
    </head>
    <body>
        <h1>Número de continentes: <span
        th:text="${continentes.length}">0</span></h1>
        <h2>Listado</h2>
```



```

        <ul>
            <li th:text="${continentes[0]}">uno</li>
        </ul>
        <ul>
            <li th:text="${continentes[4] != 'Europa'} ? 'No
es Europa' : 'Es Europa'">¿Es Europa?</li>
        </ul>
        <ul th:each="continente : ${continentes}">
            <li th:text="${continente}"></li>
        </ul>

        <h4 th:each="continente : ${continentes}"
th:text="${continente}"></h4>

        <p>Los <span
th:text="${#arrays.length(continentes)}"></span> grandes
continentes.</p>

        <p>Europa es un continente: <span
th:text="${#arrays.contains(continentes, 'Europa')}"></span>.</p>

        <p>El array de continentes está vacío <span
th:text="${#arrays.isEmpty(continentes)}"></span>.</p>

    </body>
</html>

```

La estructura **th:each** se puede usar recogiendo la variable que define a cada elemento del array en sucesivas etiquetas o en la misma etiqueta.

Estructura **th:each** en la que cada continente se muestra en una lista

```

<ul th:each="continente : ${continentes}">
    <li th:text="${continente}"></li>
</ul>

```

Estructura **th:each** en la que cada continente se muestra en un h4, misma etiqueta donde se define.

```

<h4 th:each="continente : ${continentes}"
th:text="${continente}"></h4>

```

En el código anterior también se ha mostrado un ejemplo de uso de un if de una línea, con una condición en la que si se cumple se muestra el texto tras la ? y en caso contrario el texto tras :

```
<li th:text="${continentes[4] != 'Europa'} ? 'No es Europa' : 'Es Europa' ">¿Es Europa?</li>
```

También hemos usado la función Array para obtener el número de elementos del array, si el array contiene una determinada cadena o si el array está vacío.

Thymeleaf

Thymeleaf utiliza en su dialecto estándar el lenguaje OGNL (Object-Graph Navigation Language), un lenguaje de expresiones para Java, que permite trabajar con objetos, tomando y estableciendo propiedades, haciendo llamadas a métodos, manejo de arrays, ... Sin embargo, el dialecto para Spring utiliza SpEL (Spring Expression Language), otro lenguaje de expresiones que es común a todos los módulos y tecnologías de Spring, con mismas funcionalidades.

En nuestro caso, dado que utilizaremos Thymeleaf con Spring (con su correspondiente dialecto), tendremos que hacer uso de SpEL, si bien, en la gran mayoría de los casos, las expresiones OGNL y SpEL serán iguales.

Expresiones básicas

SpEL (y OGNL) nos proveen de diferentes tipos de expresiones entre las que destacamos:

- Expresiones variables: `${...}`
- Expresiones variables de selección: `*{...}`
- Expresiones de mensaje: `#{...}`
- Expresiones de enlaces: `@{...}`
- Expresiones de fragmentos: `~{...}`

También podemos hacer uso de literales, como en otros lenguajes de expresión (textuales, numéricos, booleanos, nulos, ...)

Expresiones variables

Dada una expresión, como `${today}`, Thymeleaf buscará en el contexto una variable llamada `today`.

Tenemos también múltiples posibilidades, con la notación de punto (`${persona.nombre}`) o de corchete (`${persona['padre']['nombre']}`); así como la manipulación de maps (`${personas['Juan'].edad}`) y arrays (`${arrayPersonas[0].nombre}`).

También podemos realizar llamadas a métodos definidos en los objetos (`${persona.nombreCompleto() }`).

Expresiones de selección

Las expresiones de selección o expresiones en selección son expresiones variables que nos permiten ejecutarlas en el marco de un objeto, creando entonces expresiones más abreviadas

Los siguientes fragmentos de una plantilla son equivalentes.

```
<div th:object="${session.usuario}">
  <p>Nombre: <span th:text="*{nombre}">Luis</span>.</p>
  <p>Apellidos: <span th:text="*{apellidos}">López</span>.</p>
  <p>Nacionalidad: <span
th:text="*{nacionalidad}">Español</span>.</p>
</div>
```

```
<div>
  <p>Nombre: <span th:text="${session.usuario.nombre}">Luis
</span>.</p>
  <p>Apellidos: <span
th:text="${session.usuario.apellidos}">López</span>.</p>
  <p>Nationality: <span
th:text="${session.usuario.nacionalidad}">Español</span>.</p>
</div>
```

Thymeleaf dispone de múltiples objetos de utilidad, con decenas de métodos, que nos ayudan en las tareas más comunes:

- **#execInfo**: información sobre la plantilla que estamos procesando.
- **#messages**: tratamiento de expresiones de mensajes conjuntamente con expresiones variables.
- **#uris**: métodos para el tratamiento de URLs/URIs
- **#conversions**: métodos para ejecutar conversiones (si es que las hay configuradas)
- **#dates**: tratamiento de fechas `java.util.Date`.
- **#calendars**: tratamiento de fechas `java.util.Calendar`.
- **#numbers**: formateo de números
- **#strings**: tratamiento de cadenas de caracteres.
- **#objects**: métodos para objetos en general.
- **#booleans**: métodos para la evaluación de booleanos
- **#arrays**: métodos para el tratamiento de arrays.
- **#lists**: métodos para el tratamiento de listas.
- **#sets**: métodos para el tratamiento de sets.
- **#maps**: métodos para el tratamiento de maps.
- **#aggregates**: métodos para el cálculo de medias aritméticas y sumas en arrays o colecciones.

Operadores aritméticos

- Suma: $\{4 + 0.2\}$
- Resta: $\{3 - 0.12\}$
- Multiplicación: $\{4 * 3\}$
- División entera: $\{4 / 3\}$
- Resto: $\{5 \% 3\}$
- División no entera $\{4 / 3.0\}$
- Potencia $\{4^3\}$
- Cambio de signo $\{-(-2)\}$

Operadores textuales

- Concatenación: $\{'Hola ' + 'mundo'\}$
- Sustitución de literales: $\{Mi nombre es \{nombre\}\}$

Operadores relacionales

- Menor: $\{3 < 2\}$ o $\{3 \text{ lt } 2\}$
- Menor o igual: $\{3 \leq 3\}$ o $\{3 \text{ le } 3\}$
- Mayor: $\{3 > 2\}$ o $\{3 \text{ gt } 2\}$
- Mayor o igual: $\{3 \geq 3\}$ o $\{3 \text{ ge } 3\}$
- Igual: $\{'hola' == 'Hola'\}$ $\{'hola' \text{ eq } 'Hola'\}$
- Distinto: $\{'hola' != 'Hola'\}$ o $\{'hola' \text{ ne } 'Hola'\}$

Operadores booleanos

- And: $\{(3 < 4) \text{ and } (4 < 5)\}$
- Or: $\{(3 > 4) \text{ or } (4 < 5)\}$
- Negacion: $\{\text{not } (3 < 4) \}$

Comparaciones

th:if y th:unless

```
<div th:if="{not #lists.isEmpty(lista)}">
  <p th:text="|Nombre del producto:
  {lista[0].nombre}|">nombre</p>
</div>

<div th:unless="{not #lists.isEmpty(lista)}">
  <p>No hay productos disponibles en la lista</p>
</div>
```

Operadores condicionales

Mediante el uso del operador condición ? valor si verdadero : valor si falso

```
<p th:text="${not #lists.isEmpty(lista)} ? |Nombre del producto:
${lista[0].nombre}| : 'No hay productos disponibles en la
lista'">nombre</p>
```

Expresiones por defecto

```
<p th:text="*{descripcion}?: 'Si quiere más información sobre
nuestro productos, no dude en contactar con nosotros'">texto</p>
```

Elemento th:block (o th-block)

th:block es un contenedor de atributos Thymeleaf. Estos atributos serán procesados, y posteriormente desaparecerá el bloque, pero no su contenido. Es muy útil en varios contextos. Por ejemplo:

```
<table>
  <th:block th:each="user : ${users}">
    <tr>
      <td th:text="${user.login}">...</td>
      <td th:text="${user.name}">...</td>
    </tr>
    <tr>
      <td colspan="2" th:text="${user.address}">...</td>
    </tr>
  </th:block>
</table>
```

Iterar con Thymeleaf

Thymeleaf nos permite iterar sobre colecciones utilizando el atributo th:each.

```
<tr th:each="producto : ${productos}">
  <td th:text="${producto.nombre}">Nombre</td>
  <td th:text="${producto.precio}">1.00</td>
</tr>
```

Nos permite iterar sobre un extenso elenco de colecciones:

- java.util.List
- Arrays
- java.util.Iterable
- java.util.Collection
- java.util.Enumeration
- java.util.Iterator
- java.util.Map (devuelve objetos de tipo java.util.Map.Entry)

Estado de la iteración

Al usar `th:each`, Thymeleaf nos ofrece un buen mecanismo para identificar el estado de la iteraciones: la variable `status`:

- El índice actual (`index`), con valor inicial 0.
- El índice actual (`count`), con valor inicial 1.
- El número total de elementos (`size`).
- Si la iteración es par o impar (`even/odd`).
- Si es la primera iteración (`first`) o la última (`last`).

No vamos a profundizar mucho más en este motor de plantilla. Si se quiere profundizar más se tiene la página oficial (<https://www.thymeleaf.org/>).

Rescatar parámetros de la ruta

Los parámetros en una petición pueden ir en el path o en el query. En este apartado vamos a ver cómo rescatarlo en ambos casos.

Cuando pasamos parámetros por query están delimitados en la URL por el carácter `?`. Todo lo que hay después de este carácter es el query. En el query podemos tener parejas nombre valor separados por el carácter `&`.

Los parámetros que se pasan en el query están limitados a 4000 caracteres.

Para rescatar los parámetros del query tendremos que usar en nuestro método la anotación `@RequestParam`.

Si por el contrario queremos rescatar las variables que vienen en el path tendremos que usar la anotación `@PathVariable`.

`@RequestParam`

Vamos a ver un ejemplo de su uso en el que tendremos una petición con la siguiente url

```
http://127.0.0.1:8080/query/?nombre=Antonio
```

Nuestro método, con nombre *metodo* tendrá el siguiente aspecto para recoger el parámetro *nombre*

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class IndexController {
```

```
@GetMapping("/query")
public String metodo(@RequestParam("nombre") String name,
Model model) {
    model.addAttribute("nombre", name);
    return "mensaje";
}
```

@RequestParam recibe el parámetro nombre y lo asigna a la variable name, que es la que pasamos a nuestra plantilla para que muestre el mensaje.

Así la petición

```
http://localhost:8080/query/?name=Juan
```

da como resultado



Hola Juan

Si el parámetro tiene el mismo nombre que la variable no hay que indicarlo en el @RequestParam y el código sería el siguiente:

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class IndexController {
    @GetMapping("/query")
```



```
public String metodo(@RequestParam String nombre, Model model)
{
    model.addAttribute("nombre", nombre);
    return "mensaje";
}
}
```

¿Qué pasa si no le pasamos ningún valor a nuestra petición?

`http://localhost:8080/query`

Pues daría un error 400. De petición mal formada.

El motivo es que `@RequestParam` tiene el valor por defecto **required** a true

Para subsanar el error anterior tenemos dos opciones

- `required=false+defaultValue` en `@RequestParam`
- Uso de *Optional* de JAVA

Uso de `required=false` más `defaultValue` en `@RequestParam`. En este caso `@RequestParam` tendría más argumentos y el método `query` quedaría de la siguiente forma:

```
package dwes.java.spring.proyecto.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class IndexController {
    @GetMapping("/query")
    public String metodo(@RequestParam(name="name",
required=false, defaultValue="Mundo") String nombre, Model model)
    {
        model.addAttribute("nombre", nombre);
        return "mensaje";
    }
}
```

Si usamos el Optional de Java el resultado sería

```
package dwes.java.spring.proyecto.controllers;

import java.util.Optional;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class IndexController {
    @GetMapping("/query")
    public String metodo(@RequestParam("name") Optional<String>
name, Model model) {
        model.addAttribute("nombre", name.orElse("Mundo"));
        return "mensaje";
    }
}
```

@PathVariable

Con esta anotación recogemos las variables del propio path. Vamos a ver un ejemplo de su uso en el que tendremos una petición con la siguiente url

```
http://127.0.0.1:8080/path/Antonio
```

Nuestro método, con nombre *metodo* tendrá el siguiente aspecto para recoger el nombre *Antonio* como un parámetro del path.

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class IndexController {
    @GetMapping("/path/{nombre}")
    public String metodo(@PathVariable("nombre") String name,
Model model) {
    model.addAttribute("nombre", name);
    return "mensaje";
}
}
```

En este caso en la definición de la ruta, la variable tiene que venir encerrada entre llaves para que se considere un parámetro. Como podemos ver `PathVariable` funciona de forma parecida a `RequestParam`.

Al igual que antes con `RequestParam`, si la variable tiene el mismo nombre que el parámetro no es necesario indicarla en la anotación `PathVariable`.

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class IndexController {
    @GetMapping("/path/{nombre}")
    public String metodo(@PathVariable String nombre, Model
model) {
    model.addAttribute("nombre", nombre);
    return "mensaje";
}
}
```

Ejemplo de @RequestParam y @PathVariable con dos parámetros

```
package dwes.java.spring.proyecto.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class IndexController {

    @GetMapping("/queryMultiple")
    public String query(@RequestParam(name="nombre",
required=false, defaultValue="Mundo") String nombre,
@RequestParam(name="apellidos",
required=false,
defaultValue="apellidos") String last_name, Model model) {
        model.addAttribute("nombre", nombre);
        model.addAttribute("apellidos", last_name);
        return "mensajeMultiple";
    }

    @GetMapping("/pathMultiple/{nombre}/{apellidos}")
    public String path(@PathVariable("nombre") String name,
@PathVariable("apellidos") String last_name, Model model) {
        model.addAttribute("nombre", name);
        model.addAttribute("apellidos", last_name);
        return "mensajeMultiple";
    }
}
```

La plantilla mensajeMultiple.html quedaría así:

```
<!DOCTYPE html>
<html lang="es" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
```

```
<title>Mi primera aplicación</title>
</head>
<body>
    <h1>Hola <span th:text="${nombre}">nombre</span> <span
th:text="${apellidos}">apellidos</span></h1>
</body>
</html>
```