

CURSO DE PROGRAMACIÓN FULL STACK

# SPRING FRAMEWORK

## GUIA 1

VISTA - CONTROLADOR



## INTRODUCCION

¡Entramos en la última etapa del curso! ¡Ahora podremos incorporar todo lo que venimos aprendiendo, y darle formato a una aplicación web funcional!

Veniamos trabajando con JAVA dentro del paradigma de programación orientado a objetos (POO) que es usado por miles de millones de dispositivos, desde computadoras hasta parquímetros, pero hoy en día, uno de sus mayores usos es para el **creciente mundo de la web**.

## FUNDAMENTOS WEB

El éxito de la web se basa en dos factores fundamentales: el **protocolo HTTP** y el lenguaje de marcado HTML. El primero permite una implementación sencilla de un sistema de comunicaciones que permite enviar cualquier archivo de forma fácil, simplificando el funcionamiento del servidor y posibilitando que servidores poco potentes atiendan cientos o miles de peticiones y reduzcan de este modo los costes de despliegue. El segundo, el lenguaje HTML, proporciona un mecanismo sencillo y muy eficiente de creación de páginas enlazadas.

## ¿QUÉ ES EL PROTOCOLO HTTP?

Un protocolo es una PETICIÓN o SOLICITUD desde el cliente hacia un servidor.

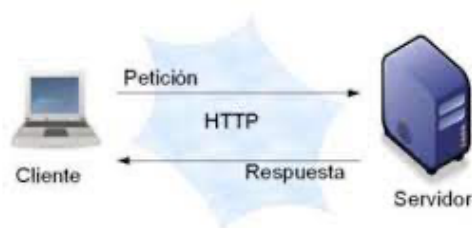
El protocolo HTTP (Hypertext Transfer Protocol) es el protocolo principal de la World Wide Web. Es un protocolo simple, orientado a conexión y sin estado. Está orientado a conexión porque emplea para su funcionamiento un protocolo de comunicaciones (TCP, o Transport Control Protocol) de modo conectado, que establece un canal de comunicaciones entre el cliente y el servidor, por el cual pasan los bytes que constituyen los datos de la transferencia, en contraposición a los protocolos denominados de datagrama (o no orientados a conexión) que dividen la serie de datos en pequeños paquetes (o datagramas) antes de enviarlos, pudiendo llegar por diversas vías del servidor al cliente.

Explicado de manera más simple, cuando escribes una dirección web en tu navegador y se abre la página que deseas, es porque tu navegador se ha comunicado con el servidor web por HTTP. Dicho de otra manera, el protocolo HTTP es el código o lenguaje en el que el navegador le comunica al servidor qué página quiere visualizar.

## HTTPS

Existe una variante de HTTP denominada HTTPS (S significa "secure", o "seguro") que utiliza el protocolo de seguridad SSL (o "Secure Socket Layer") para cifrar y autenticar el tráfico de datos, muy utilizada por los servidores web orientados al comercio electrónico o por aquellos que albergan información de tipo personal o confidencial.

### Arquitectura Cliente - Servidor



## ¿CÓMO FUNCIONA HTTP?

De forma esquemática, el funcionamiento de HTTP es como sigue: el cliente establece una conexión TCP con el servidor, hacia el puerto por defecto para el protocolo HTTP (o el indicado expresamente en la conexión), envía una orden HTTP de solicitud de un recurso (añadiendo algunas cabeceras con información) y, utilizando la misma conexión, el servidor responde enviando los datos solicitados y, además, añadiendo algunas cabeceras con información.

La manera más fácil de explicar cómo funciona HTTP es describiendo cómo se abre una página web:

1. En la barra de direcciones del navegador, el usuario teclea `example.com`.
2. El navegador envía esa *solicitud*, es decir, **la petición HTTP**, al servidor web que administre el dominio `example.com`. Normalmente, la solicitud del cliente dice algo así como “Envíame este archivo”, pero también puede ser simplemente “¿Tienes este archivo?”.
3. El servidor web recibe la *solicitud HTTP*, busca el archivo en cuestión (en nuestro ejemplo, la página de inicio de `example.com`, que corresponde al archivo `index.html`) y el servidor envía una *respuesta*. En primer lugar envía una cabecera o header. Esta cabecera le comunica al cliente, mediante un *código de estado*, el resultado de la búsqueda.
4. Si se ha encontrado el archivo solicitado y el cliente ha solicitado recibirlo (y no solo saber si existe), el servidor envía, tras el header, el message body o cuerpo del mensaje, es decir, el contenido solicitado: en nuestro ejemplo, el **archivo** `index.html`.
5. El navegador recibe el archivo y lo abre en forma de página web.

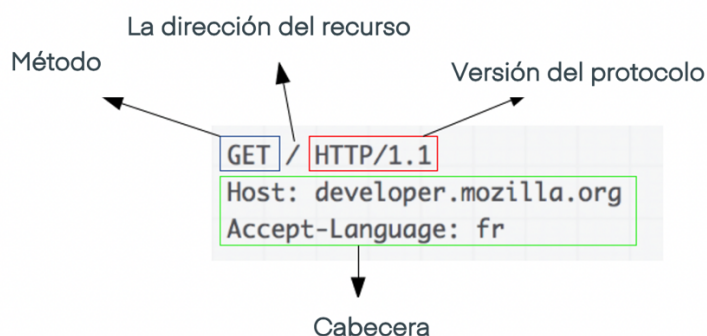
## MENSAJES HTTP

Como vimos el servidor recibe un mensaje que es la petición HTTP del usuario y después envía una respuesta HTTP al cliente/navegador, en base a la petición.

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

### Peticiones

Una petición de HTTP se ve de la siguiente manera:

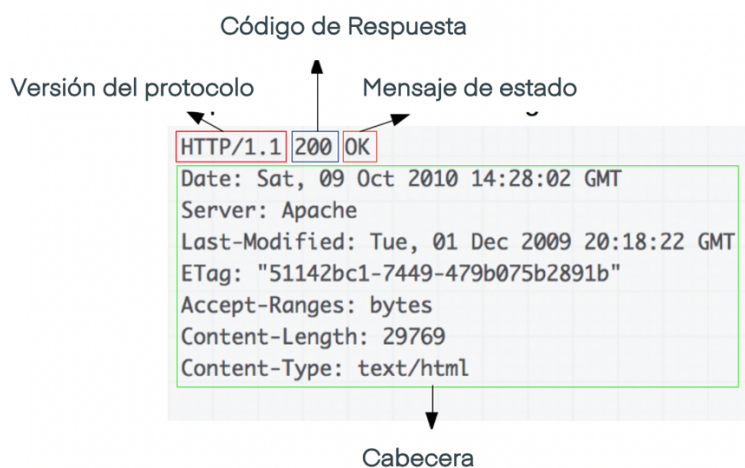


Una petición de HTTP está formada por los siguientes campos:

- **Un método HTTP**, normalmente pueden ser un verbo, como: **GET**, **POST** o un nombre como: **OPTIONS** (en-US) o **HEAD** (en-US), que defina la operación que el cliente quiera realizar. El objetivo de un cliente, suele ser una petición de recursos, usando **GET**, o presentar un valor de un formulario HTML, usando **POST**, aunque en otras ocasiones puede hacer otros tipos de peticiones.
- **La dirección del recurso pedido**; la URL del recurso, sin los elementos obvios por el contexto, como pueden ser: sin el protocolo (`http://`), el dominio (aquí `developer.mozilla.org`), o el puerto TCP (aquí el 80).
- **La versión del protocolo HTTP**.
- **Cabeceras HTTP**, opcionales, que pueden aportar información adicional a los servidores.

## Respuestas

Un ejemplo de respuesta:



Las respuestas están formadas por los siguientes campos:

- La **versión del protocolo HTTP** que están usando.
- Un **código de respuesta**, indicando si la petición ha sido exitosa, o no, y debido a qué.
- Un **mensaje de estado**, una breve descripción del código de estado.
- **Cabeceras HTTP**, como las de las peticiones.

## ¿CUÁLES SON LOS MÉTODOS DE PETICIÓN?

En la web, los clientes, como un navegador, por ejemplo, se comunican con los distintos servidores web con ayuda del protocolo HTTP, el cual regula cómo ha de formular sus peticiones el cliente y cómo ha de responder el servidor. El protocolo HTTP emplea varios métodos de petición diferentes.

### GET

**GET** es la madre de todas las peticiones de HTTP. Este método de petición existía ya en los inicios de la *world wide web* y se utiliza para **solicitar un recurso**, como un archivo HTML, **del servidor web**. Esto podría ser cuando un usuario clickea un link para ir a una página concreta.

Cuando escribes la dirección URL `www.ejemplo.com/test.html` en tu navegador, este se conecta con el servidor web y le envía una petición GET:

```
GET/test.html
```

El servidor enviaría el archivo `test.html` como respuesta.

### PARTES DE UNA URL

A la petición GET puede añadirse **más información**, con la intención de que el servidor web también la procese. Estos llamados parámetros de URL se adjuntan a la dirección URL, la URL puede estar compuesta de varias partes, y las vamos a ver a continuación:

#### Ruta (Path)

Es lo que viene **después de la barra /**.

Normalmente indica páginas y subpáginas que podemos encontrar en un sitio web.

```
www.ejemplo.com/otraPagina.html
```

#### Parámetro (Query String)

Es lo que viene **después del signo de interrogación ?**. Todos los parámetros se componen de un nombre y un valor: **“Nombre=Valor”**.

En una URL puede haber varios parámetros. Y cuando es el caso, éstos se separan con el símbolo de **ampersand &**.

Los parámetros pueden indicar diferentes cosas. A veces tienen que ver con una **búsqueda en el sitio**, a veces son parámetros de campañas publicitarias, etc.

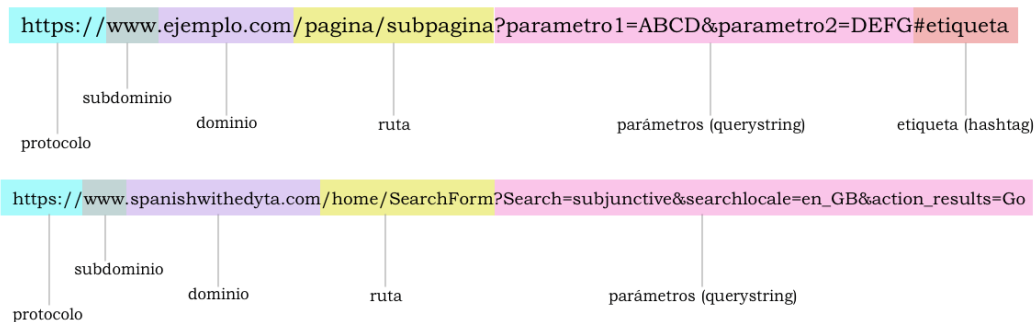
Veámoslo con este ejemplo: para buscar ciertas ofertas en la página web de una empresa de software, en la petición GET se indicará “Windows” como plataforma y “Office” como categoría:

```
GET /search?platform=Windows&category=office
```

#### Etiqueta

Las etiquetas en una URL aparecen después del hashtag #.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión. Veamos una URL completa:



## POST

Cuando se tienen que enviar al servidor web paquetes grandes de datos, como imágenes o datos de formulario de carácter privado, por ejemplo. El método GET se queda corto, porque todos los datos que se transmiten se escriben abiertos en la barra de direcciones del navegador.

En estos casos, se recurre al método POST. Este método no escribe el parámetro del URL en la dirección URL, sino que lo adjunta al encabezado HTTP.

Las peticiones POST suelen emplearse con **formularios digitales**. Abajo encontrarás el ejemplo de un formulario que recoge un nombre y una dirección de correo electrónico y lo envía al servidor por medio de POST:

```
<html>
<body>
<form action="/prueba " method="post">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<button type="submit">
</form>
</body>
</html>
```

## ¿CUÁNDO USAR UNO U OTRO?

El método POST es aconsejable cuando el usuario debe enviar datos o archivos al servidor, como, por ejemplo, cuando se rellenan formularios o se suben fotos.

El método GET es adecuado para la personalización de páginas web: el usuario puede guardar búsquedas, configuraciones de filtros y ordenaciones de listas junto al URL como marcadores, de manera que en su próxima visita la página web se mostrará según sus preferencias.

A modo de resumen:

**GET** – Utilizado para obtener un recurso del servidor, identificado por una url. Para la configuración de páginas web (filtros, ordenación, búsquedas, links, etc.).

**POST** – Utilizado para la transferencia de información y datos al servidor. Puede utilizarse para enviar parámetros y su longitud es ilimitada.

## CÓDIGOS DE RESPUESTA

Al iniciar el navegador (llamado cliente en este caso) se realiza una petición al servidor web, quien responde, a su vez, con un código de estado HTTP en forma de cadena de tres dígitos.

Con este mensaje, el servidor web le indica al navegador si su solicitud ha sido procesada correctamente, si ha ocurrido un error o si se necesita una autenticación. Como consecuencia, el código de estado HTTP se convierte en una parte esencial en la transmisión de mensajes de respuesta por parte del servidor web, que es insertado automáticamente en su encabezado. Por lo general, los usuarios se encuentran con páginas en formato HTML en vez de códigos de estado HTTP, cuando el servidor web no puede o no tiene permitido procesar la solicitud del cliente o no es posible realizar la transmisión de datos.

### TIPOS DE RESPUESTA DE LOS CÓDIGOS DE ESTADO HTTP

En principio, los códigos de estado HTTP se dividen en cinco categorías diferentes, identificadas a su vez, por el primer dígito del código. Por ejemplo, el código de estado HTTP 200 forma parte del tipo de respuesta 2xx, el código 404 del tipo de respuesta 4xx. Esta clasificación se deriva principalmente de la importancia y la función de los códigos de estado, divididos principalmente en 5 tipos:

**Códigos de estado 1xx – Información:** Cuando se envía un código de estado HTTP 1xx, el servidor le notifica al cliente que la petición actual aún continúa. Esta clase reúne y proporciona información sobre el procesamiento y envío de una solicitud.

**Códigos de estado 2xx – Éxito:** Los códigos que comienzan con un 2 informan sobre una operación exitosa. Cuando se reciben este tipo de respuestas quiere decir que la solicitud del cliente fue recibida, comprendida y aceptada. Por lo general, el usuario solo percibe la web solicitada.

**Códigos de estado 3xx – Redirecciones:** Aquellos códigos que comienzan con 3 indican que la solicitud ha sido recibida por el servidor. Sin embargo, para asegurar un procesamiento exitoso es necesario que el cliente tome una acción adicional. Este tipo de códigos aparecen principalmente cuando hay redirecciones.

**Códigos de estado 4xx – Errores del cliente:** Cuando aparece un código 4xx quiere decir que se ha presentado un error de cliente. Esto quiere decir que el servidor ha recibido la solicitud, pero esta no se puede llevar a cabo. Una de las principales causas de este tipo de respuestas son las solicitudes defectuosas. Los usuarios de Internet son informados de este error por medio de una página HTML generada automáticamente.

**Códigos de estado 5xx – Errores del servidor:** El servidor indica un error propio cuando usa un código 5xx. Este tipo de respuestas indican que la solicitud correspondiente está temporalmente deshabilitada o es imposible de llevar a cabo. De nuevo, se generará automáticamente una página en formato HTML.

### CÓDIGOS DE ESTADO HTTP MÁS IMPORTANTES

Los únicos códigos visibles para los visitantes son principalmente los códigos de error del cliente, como el 404 (Not Found), o de error del servidor como el 503 (Service Unavailable), ya que estos siempre se muestran automáticamente como páginas en formato HTML.

Pero ahora que vamos a trabajar sobre la creación de estas páginas web, va a ver códigos que nos van a informar de cosas a arreglar dentro de nuestro programa.

A continuación, presentamos una pequeña selección de los códigos de respuesta más comunes:

**200** – OK, petición procesada correctamente.

**301** – Indica al browser que visite otra dirección.

**403** – Acceso prohibido, por falta de permisos.

**404** – No encontrado, cuando el documento no existe.

**500** – Error interno en el servidor.

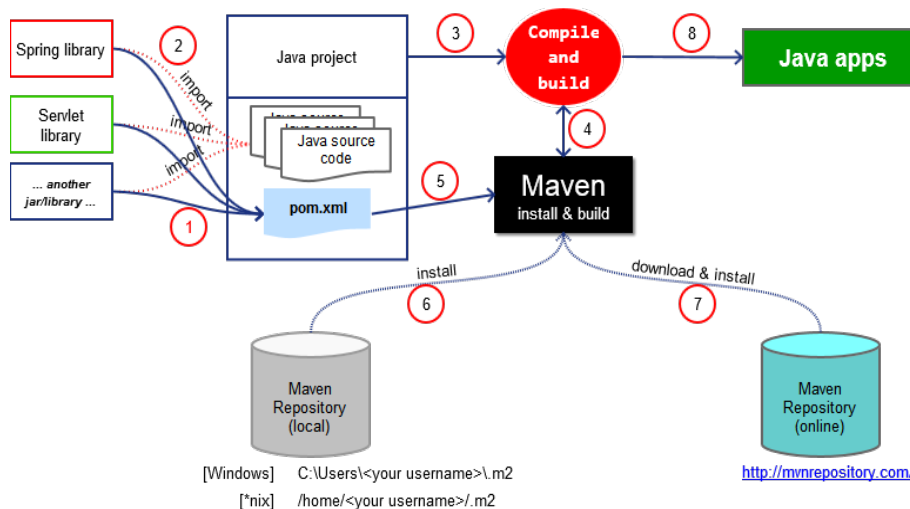
En el siguiente link podés ver todos los códigos de estado:

<https://uniwebsidad.com/tutoriales/los-codigos-de-estado-de-http?from=librosweb>

Hasta aquí vimos algunos conceptos generales del funcionamiento de los navegadores y servidores. Ahora debemos aprender a programar de tal manera de ser capaces de elaborar programas que utilicen el protocolo http y para ello debemos aprender MAVEN y SPRING:

## ¿QUÉ ES MAVEN?

Maven es una herramienta de software para la gestión y construcción de proyectos Java. Utiliza un Project Object Model (**POM**) para describir el proyecto de software a construir, sus **dependencias** de otros módulos y componentes externos, y el orden de construcción de los elementos. El modelo de configuración es simple y está basado en un formato XML (`pom.xml`). Además, Maven tiene objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. La siguiente figura ilustra los pasos que lleva a cabo esta herramienta desde la importación de librerías hasta la generación de la aplicación Java.



## DEPENDENCIAS

Uno de los puntos fuertes de Maven son las dependencias. En nuestro proyecto podemos decirle a Maven que necesitamos un jar (por ejemplo, `log4j` o el **conector de MySQL**) y maven es capaz de ir a internet, buscar esos jar y bajárselos automáticamente. Es más, si alguno de esos jar necesitara otros jar para funcionar, maven "tira del hilo" y va bajándose todos los jar que sean necesarios. Vamos a ver todo esto con un poco de detalle. Las dependencias se recopilan en el archivo `pom.xml`, dentro de una etiqueta `<dependencies>`.



Cuando ejecuta una compilación o ejecutamos un proyecto Maven, estas dependencias se resuelven y luego se cargan desde el repositorio local. Si no están presentes allí, Maven los descargará de un repositorio remoto y los almacenará en el repositorio local. También se le permite instalar manualmente las dependencias.

## AÑADIR DEPENDENCIAS EN NUESTRO PROYECTO

Para indicarle a maven que necesitamos un jar determinado, debemos editar el fichero *pom.xml* que tenemos en el directorio raíz de nuestro proyecto. En el *pom.xml* generado por defecto por maven veremos un trozo como el siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Esta es una dependencia que pone maven automáticamente cuando creamos nuestro proyecto. Presupone que vamos a usar *junit* y en concreto, la versión 3.8.1. Si nosotros queremos añadir más dependencias, debemos poner más trozos como este. Por ejemplo, si añadimos la dependencia del conector de *mysql* versión 5.1.12, debemos poner lo siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId> // CAMBIAR POR LA DEPENDENCIA
    <version>5.1.12</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

## MAVEN VS GRADLE

### ¿Qué es Gradle?

**Gradle** es una **herramienta de automatización de compilación del código abierto**, que obtuvo una rápida popularidad ya que fue diseñada fundamentalmente para construir multiproyectos, utilizando conceptos provenientes de Apache Maven.

## Diferencias Principales

A continuación, destacamos las diferencias más importantes que existen entre Maven y Gradle.

Gradle	Maven
El tiempo de construcción de Gradle es corto y rápido	El rendimiento de Maven es lento en comparación con Gradle
Los scripts de Gradle son mucho más cortos y limpios	Los scripts de Maven son un poco largos en comparación con Gradle
Utiliza lenguaje específico de dominio (DSL)	Utiliza XML
Se basa en la tarea mediante la cual se realiza el trabajo	En Maven se definen objetivos vinculados al proyecto
Admite compilaciones incrementales de la clase java	No admite compilaciones incrementales
Soporte en la mayoría de las herramientas de Integración continua	Soporte en la mayoría de las herramientas de Integración continua

## ¿QUÉ ES SPRING FRAMEWORK?

Spring es un framework alternativo al stack de tecnologías estándar en aplicaciones JavaEE. Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio.

Spring es el framework más popular para el desarrollo de aplicaciones empresariales en Java, para crear código de alto rendimiento, liviano y reutilizable. Su finalidad es estandarizar, agilizar, manejar y resolver los problemas que puedan ir surgiendo en el trayecto de la programación.

## ¿QUÉ ES UN FRAMEWORK?

Un **framework** es un entorno de trabajo que tiene como **objetivo facilitar la labor de programación** ofreciendo una serie de **características y funciones** que aceleran el proceso, reducen los errores, favorecen el trabajo colaborativo y consiguen obtener un producto de mayor calidad.

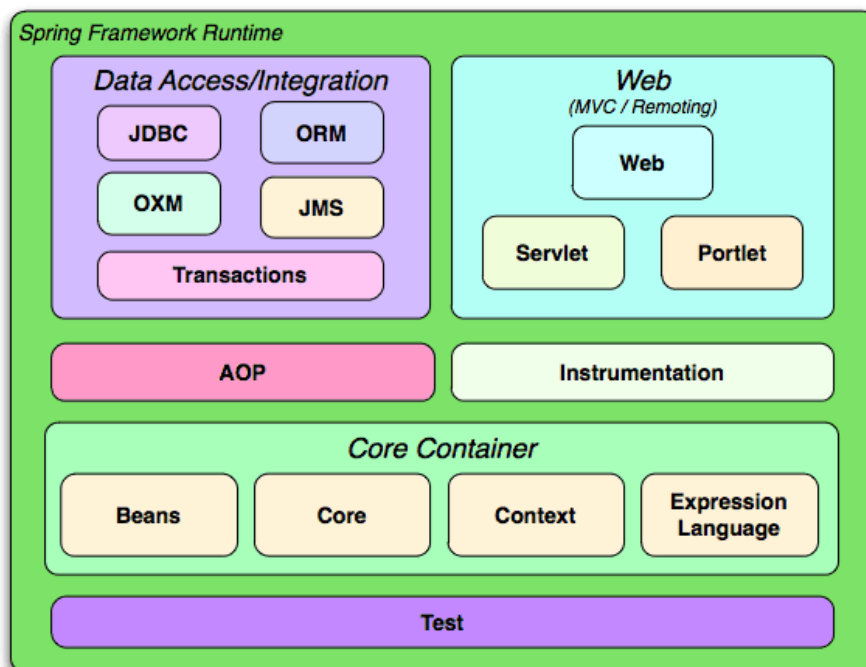
Los framework **ofrecen una estructura para el desarrollo** y no tienen que estar sujetos a un único lenguaje de programación, aunque es habitual encontrar en el mercado, distintos frameworks específicos para un lenguaje concreto.

## SPRING FUNCIONALIDADES

Spring, ofrece como elemento clave la inyección de dependencias a nuestro proyecto, pero existen otras funcionalidades también muy útiles:

- **Core container:** proporciona inyección de dependencias e inversión de control.
- **Web:** nos permite crear controladores Web, tanto de vistas **MVC** como aplicaciones REST. Esto facilita en gran medida la programación basada en **MVC (Modelo Vista Controlador)**
- **Acceso a datos:** abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (Object XML Mappers), JSM y transacciones.
- **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.



## SPRING MVC

Antes de pasar a ver la inyección de dependencias, veremos otra funcionalidad, que es el Spring MVC.

Spring Web MVC es un sub-proyecto Spring que está dirigido a facilitar y optimizar el proceso creación de aplicaciones web utilizando el patrón **Modelo Vista Controlador**.



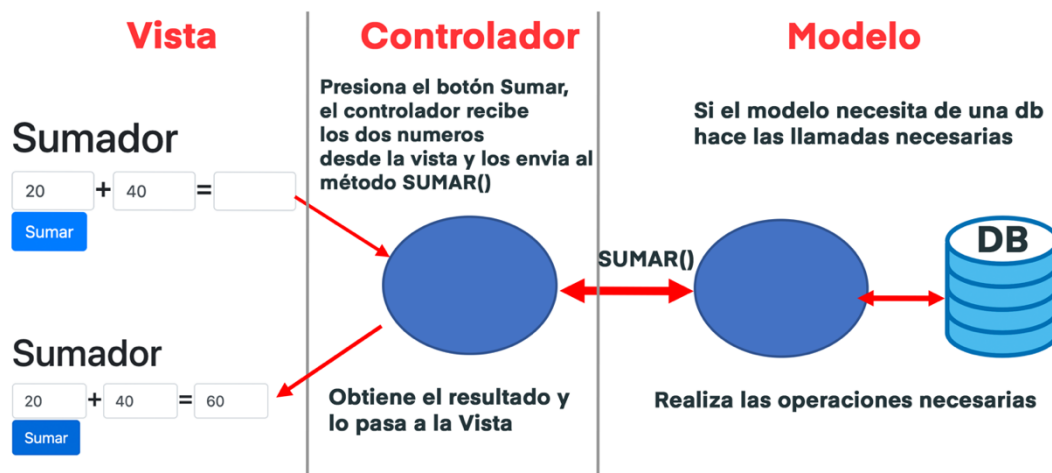
### ¿QUE ES EL PATRÓN DE DISEÑO MVC?

**MVC** es un **patrón de diseño** que se estructura mediante tres componentes: **modelo, vista y controlador**. Este patrón tiene como principio que cada uno de los componentes esté separado en diferentes objetos, esto significa que los componentes no se pueden combinar dentro de una misma clase. Sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario.

**Modelo:** Esta capa representa todo lo que tiene que ver con el acceso a datos: **guardar, actualizar, obtener datos**, además todo el código de la **lógica del negocio**, básicamente son las clases Java y parte de la lógica de negocio. No contiene ninguna lógica que describa como presentar los datos a un usuario.

**Vista:** este componente presenta los **datos del modelo al usuario**. La vista sabe cómo acceder a los datos del modelo, pero no sabe que significa esta información o que puede hacer el usuario para manipularla.

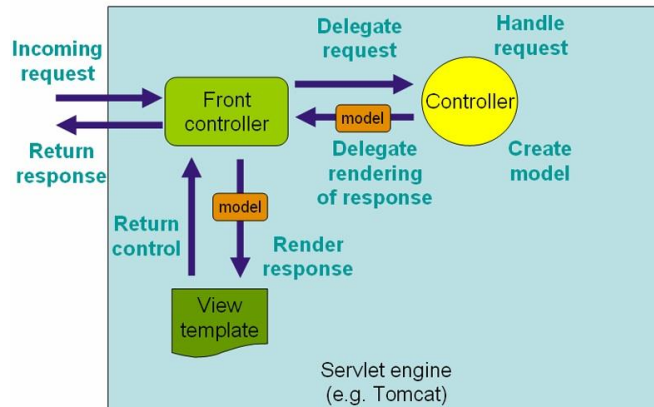
**Controlador:** este componente se encarga de gestionar las **instrucciones que se reciben, atenderlas y procesarlas**. El controlador es el encargado de **conectar el modelo con las vistas**, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encarga de direccionar al modelo la petición, recibir los resultados y entregarlos a la vista para que pueda mostrarlos.



## PROCESAMIENTO DE UNA PETICIÓN EN SPRING MVC

Spring MVC se basa en este patrón de diseño para el manejo de las peticiones http y sus respuestas.

A continuación, se describe el flujo de procesamiento típico para una petición HTTP en Spring MVC. Spring es una implementación del patrón de diseño "front controller".



- Todas las peticiones HTTP se canalizan a través del *front controller*. En casi todos los frameworks MVC que siguen este patrón, el *front controller* no es más que un servlet cuya implementación es propia del framework. En el caso de Spring, la clase *DispatcherServlet*.
- El *front controller* averigua, normalmente a partir de la URL, a qué Controller hay que llamar para servir la petición. Para esto se usa un *HandlerMapping*.
- Se llama al *Controller*, que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al *servlet*, encapsulados en un objeto del tipo *Model*. Además, se devolverá el nombre lógico de la vista a mostrar (normalmente devolviendo un String, como en JSF).
- Un *ViewResolver* se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico del paso anterior.
- Finalmente, el *front controller* (el *DispatcherServlet*) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

## INYECCIÓN DE DEPENDENCIAS

La inyección de dependencias es quizás la característica más destacable del core de Spring Framework, que consiste en que en lugar de que cada clase tenga que instanciar los objetos que necesite, **sea Spring el que inyecte esos objetos**, lo que quiere decir que es Spring el que creará los objetos y cuando una clase necesite usarlos se le pasaran (como cuando le pasas un parámetro a un método).

La **DI** (*Dependency Injector* o *Inyector de Dependencias*) consiste en que en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectarán mediante los métodos setters o mediante el constructor en el momento en el que se cree la clase y cuando se quiera usar la clase en cuestión ya estará lista, en cambio sin usar DI la clase necesita crear los objetos que necesita cada vez que se use.

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de configuración XML o mediante anotaciones.

Spring a estas clases que van a ser inyectadas por el contenedor, las llama **Spring Beans**.

## ¿QUE ES UN BEAN?

Un **Bean** es una clase de **Java** que debe cumplir los siguientes **requisitos**:

- Tener todos sus atributos privados (private).
- Tener métodos set() y get() públicos de los atributos privados que nos interese.
- Tener un constructor público por defecto.

A diferencia de los Bean convencionales que representan una clase, la particularidad de los Beans de Spring es que son objetos creados y manejados por el contenedor Spring.

## CONTENEDOR SPRING

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de **configuración XML o mediante anotaciones**. En el caso de Spring ese objeto es el contenedor IoC el cual es provisto por los módulos spring-core y spring-beans.

**Spring** se basa en el principio de **Inversión de Control (IoC)** o **Patrón Hollywood** («No nos llames, nosotros le llamaremos») consiste en:

- Un Contenedor que maneja objetos por vos, este contenedor es un archivo XML. Este archivo se llama **application-context.xml**.
- El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices vos.
- El contenedor resuelve dependencias entre los objetos que contiene.

Un ejemplo típico para ver su utilidad es el de una clase que necesita una conexión a base de datos, sin DI si varios usuarios necesitan usar esta clase se tendrán que crear múltiples conexiones a la base de datos con la consiguiente posible pérdida de rendimiento, pero usando la inyección de dependencia, las dependencias de la clase (sus atributos), son instanciados una única vez cuando se despliega la aplicación y se comparten por todas las instancias de modo que una única conexión a base de datos es compartida por múltiples peticiones.

## SPRING @CONFIGURATION

La anotación @Configuration se utiliza para la configuración basada en anotaciones de Spring. @Configuration es una anotación de marcador que indica que una clase declara uno o más beans, y puede ser procesada por el contenedor Spring para generar definiciones de beans y solicitudes de servicio para esos beans en tiempo de ejecución.

Por lo general, la clase que define el método Main es un buen candidato como principal @Configuration.

De todas formas, recomendamos utilizar la anotación @Configuration en una clase exclusiva de configuraciones, donde también tendremos las configuraciones de seguridad de nuestro proyecto.

```
import org.springframework.context.annotation.Configuration;

@Configuration

public class Configuraciones {

}
```

## ANOTACIONES

Ahora, cuando una clase está anotada con una de las siguientes anotaciones, Spring las registrará automáticamente en el application-context. Esto hace que la clase esté disponible para la inyección de dependencias en otras clases y esto se vuelve vital para construir nuestras aplicaciones. Estas anotaciones se las conoce como **Spring Stereotypes** y se pueden encontrar en el paquete **org.springframework.stereotype**.

### SPRING STEREOTYPES

Existen varios tipos de anotaciones Spring Stereotypes. Nos centraremos en las dos principales.

**@Controller:** Este estereotipo realiza las tareas de controlador y gestión de la comunicación entre el usuario y la aplicación. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas. Donde se realiza la asignación de solicitudes desde la página de presentación, es decir, la capa de presentación (o Interface) no va a ningún otro archivo, va directamente a la clase **@Controller** y comprueba la ruta solicitada en la anotación **@RequestMapping** escrita antes de las llamadas al método si es necesario.

@Controller

```
public class Controlador{}
```

Esta es una clase de controlador simple que contiene métodos para manejar peticiones HTTP para diferentes URLs.

**@Autowired** Esta anotación le indica a Spring dónde debe ocurrir una inyección. Si se lo coloca en un método, por ejemplo: setMovie, entiende (por el prefijo que establece la anotación @Autowired) que se necesita inyectar un bean. Spring busca un bean de tipo Movie y, si lo encuentra, lo inyecta a este método. Sustituye la declaración de los atributos del bean en el xml. @Autowired se emplea para generar la inyección de dependencia de un tipo de Objeto que pertenece a una clase con la @Component(@Controller, @Service, @Repository)

@Autowired

```
private final PeliculaServicio peliculaServicio;
```

## SPRING MVC ANOTACIONES

También existen otras anotaciones que nos ayudarán con el manejo del patrón de diseño Spring MVC. Nos darán facilidades para la comunicación entre las vistas, el controlador y los modelos.

**@Controller:** esta anotación se repite en este apartado, ya que nos da la posibilidad de marcar a una clase como un controlador. Esta anotación se utiliza para crear una clase como controlador web, que puede manejar las solicitudes del cliente y enviar una respuesta al cliente.

**@RequestMapping:** La clase Controller contiene varios métodos para manejar diferentes peticiones HTTP, pero ¿cómo asigna Spring una petición en particular a un método del controlador en particular? Bueno, eso se hace con la ayuda de la anotación **@RequestMapping**. Es una anotación que se especifica sobre un método del controlador.

Proporciona el mapeo entre la **ruta de la petición** y el **método del controlador**. También admite alguna opción avanzada que se puede usar para especificar métodos de controlador separados para diferentes tipos de petición en el mismo URI como puede especificar un método para manejar una petición GET y otro para manejar la petición POST.

```
@Controller
public class Controlador{

@RequestMapping("/")
    public String hola(){
        return "Hola Spring MVC";
    }
}
```

En este ejemplo, la página de inicio se asignará a este método de controlador. Entonces, cualquier petición sobre la ruta localhost:8080 "/", irá a este método que devolverá "Hola Spring MVC".

**@GetMapping:** esta anotación se utiliza para asignar solicitudes HTTP GET a métodos de controlador específicos. **@GetMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.GET).

```
@Controller
public class Controlador{

@GetMapping("/")
    public String hola(){
        return "Hola Spring MVC";
    }
}
```

**@PostMapping:** esta anotación se utiliza para asignar solicitudes HTTP POST a métodos de controlador específicos. **@PostMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.POST).

```
@Controller
public class Controlador{

@PostMapping("/guardar")
    public String guardarUsuario(){
        return "Usuario Guardado ";
    }
}
```

**@RequestParam:** esta es otra anotación Spring MVC útil que se usa para vincular los parámetros de una petición HTTP a los argumentos de un método controlador. Por ejemplo, si envía parámetros de un formulario junto con URL para guardar un usuario, el método puede obtenerlos como argumentos propios.

```
@GetMapping("/libro"){
    public void mostrarDetalleLibro(@RequestParam("ISBN") String ISBN){
        System.out.println(ISBN);
    }
}
```

Si accedes a tu aplicación web que proporciona detalles del libro con un parámetro de consulta(query string) como el siguiente:

http://localhost:8080/libro?ISBN=900848893

Entonces se llamará al método del controlador porque está vinculado a la URL "/libro" y el **parámetro de consulta ISBN** se usará para completar el **argumento del método** con el mismo nombre "ISBN" dentro del método **mostrarDetalleLibro()**.



De esa manera podemos obtener en nuestro controlador un dato que viaja a través de una URL, que va a venir de una petición HTTP.

**@PathVariable:** esta es otra anotación que se utiliza para recuperar datos de la URL. A diferencia de la anotación @RequestParam que se usa para extraer parámetros de consulta, esta anotación permite al controlador manejar una petición HTTP con URLs parametrizadas, estas serían URLs que tiene parámetros como parte de su ruta, por ejemplo:

http://localhost:8080/libro/900848893

Entonces para poder acceder a este detalle que se encuentra en la ruta de la URL, usaríamos la anotación @PathVariable de la siguiente manera:

```
@GetMapping("/libro/{ISBN}") {  
    public void mostrarDetalleLibro(@PathVariable("ISBN") String ISBN){  
        System.out.println(ISBN);  
    }  
}
```

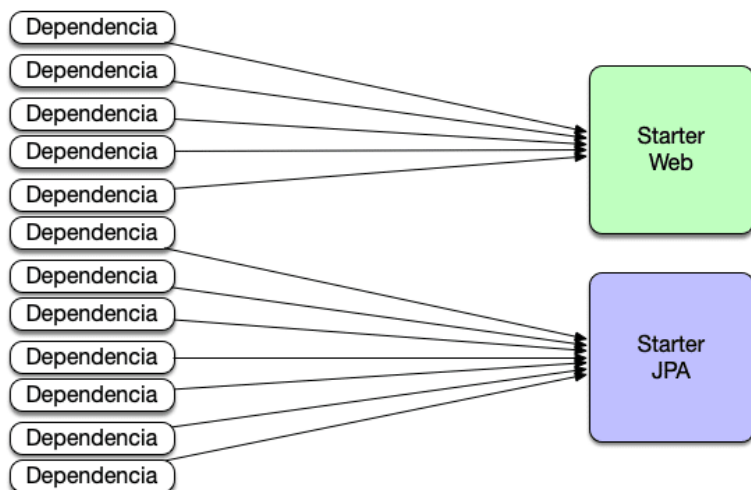
La variable Path o variable de ruta se representa entre llaves como {ISBN} en nuestra ruta de petición, lo que significa que la parte después de /libro se extrae y se completa en el ISBN del argumento del método, que está anotado con @PathVariable.

## SPRING BOOT

**Spring Boot** es una de las tecnologías dentro del mundo de Spring de las que más se está hablando últimamente. **¿Qué es y cómo funciona Spring Boot?** Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con Spring Framework

- 1 seleccionar jars con maven
- 2 crear la aplicación
- 3 desplegar en servidor

**Fundamentalmente existen tres pasos a realizar.** El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar desarrollamos la aplicación y en tercer lugar la desplegamos en un servidor. Si nos ponemos a pensar un poco a detalle en el tema, **únicamente el paso dos es una tarea de desarrollo.** Los otros pasos están más orientados a infraestructura. No deberíamos tener que estar eligiendo continuamente las dependencias y el servidor de despliegue.



## SPRING INITIALIZER

SpringBoot nace con la intención de simplificar los pasos 1 y 3 y que nos podamos centrar en el desarrollo de nuestra aplicación. ¿Cómo funciona?. El enfoque es sencillo y lo entenderemos realizando un ejemplo. Para ello nos vamos a conectar al asistente de Boot que se denomina Spring Initializer.

The screenshot shows the Spring Initializer web application interface. It has a dark theme with green accents. The interface 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 various versions: '2.6.0 (SNAPSHOT)', '2.6.0 (M2)', '2.5.6 (SNAPSHOT)', '2.5.5' (selected), and '2.4.12 (SNAPSHOT)'. There is also a '2.4.11' option.
- Project Metadata:** Includes input fields for 'Group' (com.ejemplospring), 'Artifact' (EjemploGuia), 'Name' (EjemploGuia), 'Description' (Proyecto de ejemplo), and 'Package name' (com.ejemplospring.EjemploGuia). It also has a 'Packaging' section with 'Jar' (selected) and 'War' options, and a 'Java' version section with '17', '11', and '8' (selected) options.
- Dependencies:** A section on the right with a button 'ADD DEPENDENCIES... + B'. It lists several dependencies:
  - Spring Boot DevTools:** Labeled 'DEVELOPER TOOLS'. Description: 'Provides fast application restarts, LiveReload, and configurations for enhanced development experience.'
  - Spring Web:** Labeled 'WEB'. Description: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.'
  - Thymeleaf:** Labeled 'TEMPLATE ENGINES'. Description: '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.'
  - MySQL Driver:** Labeled 'SQL'. Description: 'MySQL JDBC and R2DBC driver.'
  - Spring Data JPA:** Labeled 'SQL'. Description: 'Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.'

At the bottom, there are three buttons: 'GENERATE ⌘ + ↵', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

En este caso voy a construir una aplicación **Spring MVC** y elijo la dependencia web o **Spring Web**. Pulsamos generar proyecto y nos descargará un proyecto Maven en formato zip. Lo descomprimos y lo abrimos en nuestro IDE, cuando lo vayamos a compilar, Maven se encargará de descargar todas las dependencias y sumarlas a nuestro proyecto.

Una vez que se termine de descargar nuestro proyecto Maven, se convertirá en proyecto Spring para poder trabajar, dentro encontraremos la clase **EjemploGuiaApplication**, se verá de la siguiente manera:

```

1. package com.ejemplospring;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5.
6. @SpringBootApplication
7. public class EjemploGuiaApplication{
8.
9.     public static void main(String[] args) {
10.         SpringApplication.run(EjemploGuiaApplication.class, args);
11.     }
12. }

```

Esta clase es la encargada de arrancar nuestra aplicación de Spring a diferencia de un enfoque clásico no hace falta desplegarla en un servidor web ya que Spring Boot provee de uno. Vamos a modificarla y añadir una anotación.

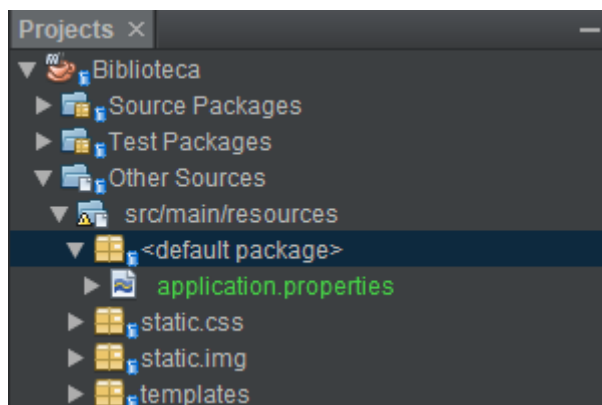
## MANOS A LA OBRA

¡A continuación, invitamos a que creen su **primer proyecto Spring** desde Spring Initializr!

## APPLICATION.PROPERTIES

Spring Framework trae incorporado un mecanismo para la configuración de aplicaciones usando un archivo llamado **application.properties**.

Este archivo se encuentra dentro de la carpeta `src/main/resources/<default package>`, como se muestra en la siguiente figura.



Este archivo nos permite ejecutar una aplicación en un entorno diferente.

En resumen, podemos usar el archivo `application.properties` principalmente con dos objetivos:

- Configurar el marco Spring Boot.
- Definir propiedades de configuración personalizadas para nuestra aplicación.

## COMO EDITAR APPLICATION.PROPERTIES

Abrimos el archivo application.properties, que inicialmente lo encontraremos vacío, y pegamos el siguiente texto dentro del mismo.

```
spring.datasource.url: jdbc:mysql://localhost:3306/DATABASE?allowPublicKeyRetrieval=true&useSSL=false&useTimezone=true&serverTimezone=GMT&characterEncoding=UTF-8
```

```
spring.datasource.username: root
```

```
spring.datasource.password: root
```

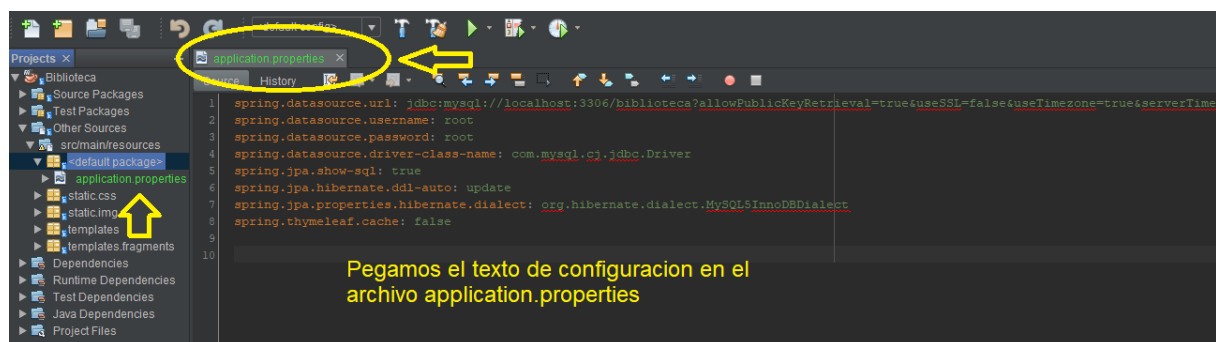
```
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
```

```
spring.jpa.show-sql: true
```

```
spring.jpa.hibernate.ddl-auto: update
```

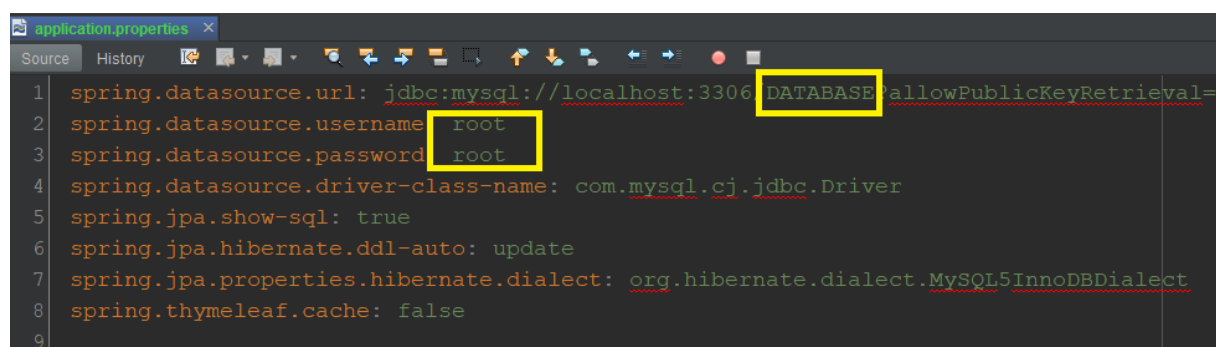
```
spring.jpa.properties.hibernate.dialect:  
org.hibernate.dialect.MySQL5InnoDBDialect
```

```
spring.thymeleaf.cache: false
```



En la primer linea, podemos ver el nombre de la Base de Datos a la que nos vamos a conectar.

En las líneas 2 y 3 encontramos las credenciales de MySQL (usuario y contraseña).



Para más información sobre el resto de las configuraciones iniciales, te invitamos a investigar este link: <https://www.javatpoint.com/spring-boot-properties>.

## SERVIDOR LOCAL

Levantar un servidor o tener un servidor a nuestra disposición no es algo fácil, ni barato. Por suerte Spring Boot nos deja, a través de **Apache Tomcat** y la clase que vimos previamente levantar un servidor local.

Tomcat nos permite hacer una conexión por red a si mismo, o escuchar a la espera de conexiones entrantes que se vayan a originar en el mismo dispositivo.

Se usa para desarrollo y pruebas: normalmente como desarrollador montas un servidor web (apache) y este escucha en el puerto 8080. Entonces lo que hace el desarrollador para probar las páginas web que está creando, o las aplicaciones web, o los APIs o servicios, es apuntar su navegador a <http://localhost/> o <http://localhost:8080/> para hacer sus pruebas, cuando el puerto 80 se usa no se requiere especificar, solo cuando es un puerto diferente se tiene que poner con “:” después del nombre

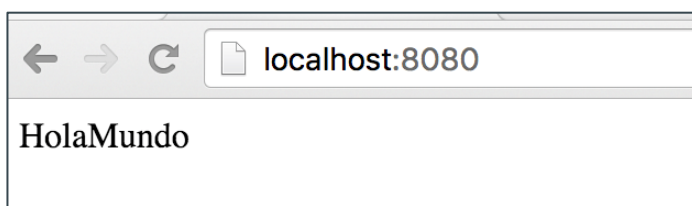
Por lo que si tenemos la siguiente clase:

`@Controller`

```
public class ControladorHola {  
    @GetMapping("/")  
    Public String home() {  
        return "holaMundo";  
    }  
}
```

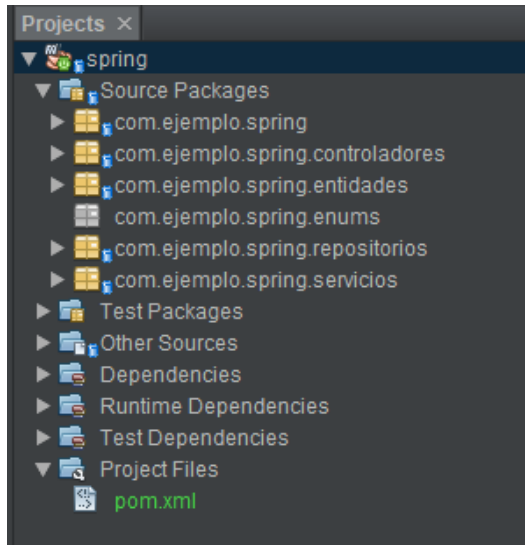
El controlador recibe la petición GET de HTTP con el GetMapping y usando el **return**, retorna como respuesta HTTP una pagina HTML como String, con el nombre holaMundo, que dentro tiene un **<p>HolaMundo</p>**, también por eso ponemos el método como String.

Entendido esto, vamos a nuestra clase EjemploGuiaApplication y corremos nuestro proyecto, se nos va a levantar un servidor local, por lo que si vamos a **localhost:8080/**, nos encontraremos con la siguiente página:



## PROGRAMACIÓN EN CAPAS

La programación por capas es un estilo de programación en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño.



A continuación, tendremos un modelo de Entidad, en base a la cual crearemos las capas del proyecto:

```
@Entity
public class Entidad {

    @Id
    @GeneratedValue(generator = "uuid")
    private String id;
    private String nombre;
    private Integer edad;

    @Temporal(TemporalType.DATE)
    private Date fecha;

    @Enumerated(EnumType.STRING)
    private Rol rol;
```

### CLASE ENTITY

En esta clase podemos ver distintas anotaciones correspondientes a Spring y a JPA.

1. **@GeneratedValue** nos permite generar un id de forma automática.

La estrategia de generación `generator="uuid"` crea un valor String alfanumérico aleatorio. Esta estrategia es la recomendada a la hora de generar valores de identificadores únicos formato String.

En caso de querer generar un id tipo de dato numérico y autoincremental, podemos utilizar la anotación `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

2. **@Temporal** nos permite mapear las fechas con la base de datos de una forma simple. (TemporalType.DATE) hace referencia al tipo de dato que trabajaremos desde la base de datos.
3. **@Enumerated** nos permite mapear un objeto de tipo Enum. La aclaración (EnumType.STRING) indica el tipo de dato que compone a ese Enum.

## CAPA DE INTERFAZ (FRONT)

Esta capa resuelve la presentación de datos al usuario. Esta capa se encarga de “dibujar” las pantallas de la aplicación al usuario, y tomar los eventos que el cliente genere (por ejemplo, el hacer click en un botón).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo HTML</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div>Capa de Interfaz</div>
  </body>
</html>
```

## CAPA DE COMUNICACIÓN

En esta capa están los controladores y es capa responsable de mediar entre la interfaz de usuario y las capas inferiores. En esta capa contiene el dispatcher encargado de enrutar las peticiones, así como los controladores de acceso a los servicios web.

```
@Controller
@RequestMapping("/")
public class EntidadControlador {

    @GetMapping("/home")
    public String home () {
        return "index.html";
    }
}
```

## CAPA DE SERVICIOS

Esta capa resuelve la lógica de la aplicación. Contiene los algoritmos, validaciones y coordinación necesaria para resolver la problemática. Los elementos fundamentales de esta capa son los objetos de dominio. Estos objetos representan los objetos principales del negocio. La lógica para manipular los objetos que representan los datos se encuentra en los llamados objetos de negocio (Service Object).

```

@Service
public class EntidadServicio {

    @Autowired
    private EntidadRepositorio entidadRepositorio;

    @Transactional()
    public Entidad guardar(String nombre,int edad, Rol rol) throws Exception {

        Entidad entidad = new Entidad();

        entidad.setNombre(nombre);
        entidad.setEdad(edad);
        entidad.setFecha(new Date());
        entidad.setRol(rol);

        // persisto el objeto en la base de datos con la funcion save de la clase Repository.
        return entidadRepositorio.save(entidad);
    }
}

```

- **@Transactional** es una anotación que debemos utilizar cada vez que vayamos a hacer una "transacción" en la DB. (una modificación persistente).  
En caso de querer realizar simplemente una consulta, podemos mapear a través de la misma anotación, pero indicándole que será solo de lectura:  
**@Transactional(readonly=true)**.

## CAPA DE ACCESO A DATOS (REPOSITORIOS)

Esta capa resuelve el acceso a datos, abstrayendo a su capa superior de la complejidad del acceso e interacción con los diferentes orígenes de datos. Esta capa se encarga de proveer un API simple de usar, orientado al negocio, sin exponer complejidades propias de un repositorio de datos.

En esta capa se resuelven:

- cualquier acceso a la base de datos
- cualquier acceso a filesystem
- cualquier acceso a otros sistemas
- cualquier acceso a un repositorio de datos en cualquier forma.

Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para nuestras clases de negocio se simplifica porque solo vamos a necesitar definir la interfaz.

Necesariamente hay que declarar la clase **Entidad** con la que trabajará el repositorio y el **tipo de dato del id** de la entidad nombrada.

```

@Repository
public interface EntidadRepositorio extends JpaRepository<Entidad, String>{

}

```

Spring Data nos provee de muchos métodos de consulta con sólo declarar esta clase. Algunos ejemplos son: count, delete, deleteAll, deleteAll, deleteAllById, deleteById, existsById, findById, save.



Recomendamos Investigar el siguiente link donde podrán encontrar las características de cada método según la documentación oficial de Spring:

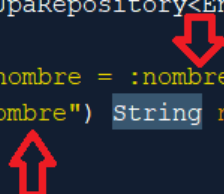
- <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

Si los métodos anteriores no nos bastan, podemos declarar métodos de búsqueda propios en la interfaz, siguiendo una convención de nomenclatura jpql podremos realizar el siguiente tipo de consultas:

```
@Repository
public interface EntidadRepositorio extends JpaRepository<Entidad, String> {

    @Query("SELECT e FROM Entidad e WHERE e.nombre = :nombre")
    public Entidad buscarPorNombre(@Param("nombre") String nombre);

}
```



En esta @Query personalizada, podemos realizar una búsqueda a través de los parámetros señalados.

La anotación @Query es necesaria para que Spring sepa que tiene que evaluarlo como una consulta a base de datos.

El parámetro ":nombre" hace referencia al señalado en la anotación @Param("nombre"), seguido por el tipo de dato a evaluar.

## THYMELEAF

Thymeleaf es un motor de plantillas, es decir, es una tecnología que nos va a permitir definir una plantilla y, junto con un modelo de datos, obtener un nuevo documento, sobre todo en entornos web.

Para saber más sobre Thymeleaf, recomendamos meterse en su documentación de Thymeleaf:

- <https://www.thymeleaf.org/>

## ¿QUE ES EXACTAMENTE UN MOTOR DE PLANTILLAS?

El motor de plantillas (utilizado específicamente aquí para el desarrollo web) se genera para separar la interfaz de usuario (Vistas), de los datos comerciales (Modelos), puede generar documentos en un formato específico y el motor de plantillas para el sitio web generará un estándar Documento HTML.

Las plantillas, o más exactamente los motores de plantillas (templates engines) leen un fichero de texto, que contiene la presentación ya preparada en HTML, e inserta en él la información dinámica que le ordena el Controlador, la parte que une la vista con la información.

Veamos un ejemplo para ver las posibilidades de las plantillas, que no acaban, ni mucho menos, en la web. La sintaxis a utilizar depende del motor de plantillas utilizado, pero todos son muy similares. Los motores de plantillas suelen tener un pequeño lenguaje de script que permite generar código dinámico, como listas o cierto comportamiento condicional, pero esto también depende del lenguaje.

Este lenguaje de script es absolutamente mínimo, lo justo para posibilitar ese comportamiento dinámico:

```
<html>
<body>
Hola ${nombre}
</body>
</html>
```

Está claro que este ejemplo, que es una pequeña variación del famoso "Hola Mundo", es bastante simple. Lo que está sucediendo, al procesar este fichero, el motor de plantillas lo recorrerá, analizará y sustituirá esa “*etiqueta clave*” **`\${nombre}** por el texto que le hallamos indicado, el nombre del visitante, por ejemplo, de forma que tengamos una presentación personalizada.

Básicamente, el motor de plantillas se encarga de recibir, una variable de tipo String llamada nombre, que se la va a enviar el controlador a la vista y la hará parte del HTML, haciéndolo dinámico. Por lo que los diferentes usuarios verán diferentes resultados.

## VENTAJAS THYMELEAF

**Permite realizar tareas que se conocen como natural templating.** Es decir, como está basada en añadir atributos y etiquetas, sobre todo HTML, va a permitir que nuestras plantillas se puedan renderizar en local, y esa misma plantilla después utilizarla también para que sea procesada dentro del motor de plantillas. Por lo cual **las tareas de diseño y programación se pueden llevar conjuntamente.**

## TIPOS DE EXPRESIONES

Permite trabajar con varios tipos de expresiones:

**Expresiones variables:** Son quizás las más utilizadas, como por ejemplo `${...}`

**Expresiones de selección:** Son expresiones que nos permiten reducir la longitud de la expresión si prefijamos un objeto mediante una expresión variable, como por ejemplo `*{...}`

**Expresiones de enlace:** Nos permiten crear URL que pueden tener parámetros o variables, como por ejemplo `@{...}`

## EXPRESIONES VARIABLES

Algunos ejemplos de expresiones variables son:

Podemos usar la notación de puntos para acceder a las propiedades de un objeto.

```
${sesión.usuario.nombre}
```

Uno de los atributos que podemos usar es **th:text** con diferentes etiquetas HTML, para poder mostrar, por ejemplo, el nombre del autor de un libro. También podemos navegar entre objetos.

```
<span th:text="${libro.autor.nombre}">
```

También podemos llamar a métodos definidos en nuestros propios objetos, lo vamos a poder hacer desde las plantillas.

```
<td th:text="${myObject.myMethod()}">
```

## ATRIBUTOS BÁSICOS

Los atributos básicos más conocidos con los que nos podemos encontrar son:

### TH:TEXT

**th:text:** Permite reemplazar el texto de la etiqueta por el valor de la expresión que le demos.

```
<p th:text="${saludo}">saludo</p>
```

### TH:EACH

**th:each:** Nos va a permitir repetir tantas veces como se indique o iterar sobre los elementos de una colección.

```
<li th:each="libro : ${libros}"  
th:text="${libro.titulo}">El Quijote</li>
```

La plantilla recibe la colección libros, y crea una variable llamada libro que va a ser en algún momento todos los elementos de nuestra colección, al igual que el for each de Java. Después, usamos la variable libro para acceder solo al título y al th:text para mostrar en el HTML el título en cuestión.

### TH:VALUE

En la guía de HTML cuando estudiamos los inputs, aprendimos que a los inputs en algunos casos puede resultarnos interesante asignar un valor definido al campo en cuestión.

Este valor inicial del campo podría ser expresado mediante el atributo **value**. Thymeleaf nos ofrece hacer esto, pero de manera dinámica, con el atributo **th:value**, para darle a los inputs valores iniciales distintos, dependiendo de que envíe el controlador.

```
<input type="text" name="instituto" th:value="${nombreInstituto}">
```

Esto nos haría pensar que es el mismo atributo que th:text, pero no, ya que th:text nos permite darle un valor a cualquier etiqueta de texto, mientras que th:value solo sirve para las etiquetas input.

### TH:IF

A veces, vamos a necesitar que un **fragmento de nuestra plantilla** solo aparezca cuando se cumple una **determinada condición**.

Los atributos **th:if** y **th:unless**, nos permiten mostrar un elemento de HTML dependiendo del resultado de una determinada condición.

```
<td>  
  <span th:if="${persona.sexo == 'F'}">Femenino</span>  
  <span th:unless="${persona.sexo == 'M'}">Masculino</span>  
</td>
```

Si el valor de persona.sexo es igual a F, entonces el elemento span va a mostrar la palabra **Femenino**.

En cambio, si el valor es M, entonces el elemento muestra la palabra **Masculino**.

## TH:HREF

Sirve para construir URLs que podemos utilizar en cualquier tipo de contexto.

Podríamos utilizarlas para hacer enlaces para URLs que sean absolutas o relativas al propio contexto de la aplicación, al servidor, al documento, etc.

Estos son unos ejemplos:

```
<a th:href="@{order/details}">...</a>
<a th:href="@{.../documents/report}">...</a>
<a th:href="@{http://www.micom.es/index}">...</a>
```

## MODEL MAP

Ya vimos cómo, gracias a Thymeleaf podemos recibir del controlador una variable y mostrarla en nuestro HTML, pero, ¿cómo hacemos para enviar esa variable desde nuestro controlador a nuestro HTML?

Para resolver este problema vamos a utilizar el objeto **ModelMap**, este objeto es parte del paquete **org.springframework.ui.ModelMap**.

El objeto ModelMap tiene el método **addAttribute** que nos permite enviar variables nuestro HTML, la ventaja del objeto ModelMap, es que también nos permite enviar Colecciones a nuestro HTML.

El método `addAttribute(String variable, Objeto nombreObjeto)`, recibe dos argumentos, una es variable de tipo String, que va ser el identificador que le vamos a poner al objeto o colección, que es el identificador con el que va a viajar al HTML y que va a tener que coincidir con la variable de Thymeleaf, y la otra es el objeto de Java que queremos mandar al HTML .

Pongamos un ejemplo, supongamos que tenemos la siguiente etiqueta en HTML con Thymeleaf, en el `th:text`, decimos que va a recibir una variable llamada nombre:

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

En el controlador tendremos el siguiente método:

```
@Controller
```

```
public class ControladorHola {
    @GetMapping("/")
    String home(ModelMap model) {
        String nombre = "Fernando"
        model.addAttribute("nombre", nombre)
        return "paginaHTML";
    }
}
```

Como podemos ver en el controlador, recibimos como argumento un ModelMap, esto es para que podemos recibir cualquier modelo que venga de la petición y para que podamos enviar los modelos que queramos como respuesta de x peticiones.

Usando `model.addAttribute()`, pasamos dos cosas, uno el identificador con el que el objeto va a viajar a la vista, que es **"nombre"**, recordemos que tiene que coincidir con la variable de Thymeleaf, y dos pasamos el objeto en sí.

Entonces cuando se llame a este controlador en localhost:8080/, se enviará el modelo “nombre”, lo recibirá la vista, gracias a Thymeleaf y mostrará el nombre Fernando.

## MODEL Y MODELANDVIEW

Además del ModelMap, también podemos encontrarnos con Model o con Model And View. Si bien en el curso recomendamos utilizar ModelMap, a continuación, dejamos una breve descripción de estas herramientas.

### MODEL

Es una Interfaz. Define un contenedor para los atributos del modelo y está diseñado principalmente para agregar atributos al modelo.

Ejemplo:

```
@GetMapping("/")
public String imprimirHola(Model model) {
    model.addAttribute("mensaje", "¡¡¡Hola mundo!!!");
    return "hola";
}
```

### MODELANDVIEW

ModelAndView es un objeto que contiene tanto el modelo como la vista. El controlador devuelve el objeto ModelAndView y DispatcherServlet resuelve la vista utilizando View Resolvers y View.

La vista es un objeto que contiene el nombre de la vista en forma de cadena y el modelo es un mapa para agregar varios objetos.

Ejemplo:

```
@GetMapping("/")
public ModelAndView helloWorld() {
    String message = "Hello World!";
    return new ModelAndView("welcome", "message", message);
}
```

## EJERCICIOS DE APRENDIZAJE

Para la realización de este trabajo práctico **se recomienda ver todos los videos de Spring**, de esta manera sabemos todos lo que tenemos que hacer, antes de empezar a hacerlo.



**VIDEOS:** Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

### 1- EGG NEWS

Vamos a crear un sitio de Noticias llamado EggNews.

Para ello necesitaremos crear una entidad llamada **Noticia**. Crearemos los servicios, repositorios y controladores necesarios para que todo funcione.

- Necesitamos que la entidad Noticia tenga Título, cuerpo y una foto.
- Por ahora el Usuario podrá crear, modificar y eliminar noticias.

#### **Desarrollaremos:**

- Vista inicio, donde se encuentren tarjetas(bootstrap) con el título y la foto de cada noticia, ordenadas de más reciente a más antigua.
- Vista noticia, donde a través de la tarjeta del inicio accedemos a la noticia completa.
- Vista panelAdmin donde gestionaremos las noticias.

## BIBLIOGRAFÍA

- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/protocolo-http/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/http-request/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/una-mirada-a-los-codigos-de-estado-http-mas-comunes/>
- <https://edytapukocz.com/url-partes-ejemplos-facil/>
- <https://howtodoinjava.com/maven/maven-dependency-management/>
- [http://chuwiki.chuidiang.org/index.php?title=Dependencias\\_con\\_maven](http://chuwiki.chuidiang.org/index.php?title=Dependencias_con_maven)
- <https://www.arquitecturajava.com/que-es-spring-framework/>
- <https://programandointentandolo.com/2013/05/inyeccion-de-dependencias-en-spring.html>
- <https://proitcsolution.com.ve/inyeccion-de-dependencias-spring/>
- <https://www.java67.com/2019/04/top-10-spring-mvc-and-rest-annotations-examples-java.html>
- <https://www.danvega.dev/blog/2017/03/27/spring-stereotype-annotations/>
- <https://www.arquitecturajava.com/spring-stereotypes/>
- <https://www.arquitecturajava.com/que-es-spring-boot/>
- <https://openwebinars.net/blog/que-es-thymeleaf/>