

UF5

Introducción, Spring y creación de servicios **REST**

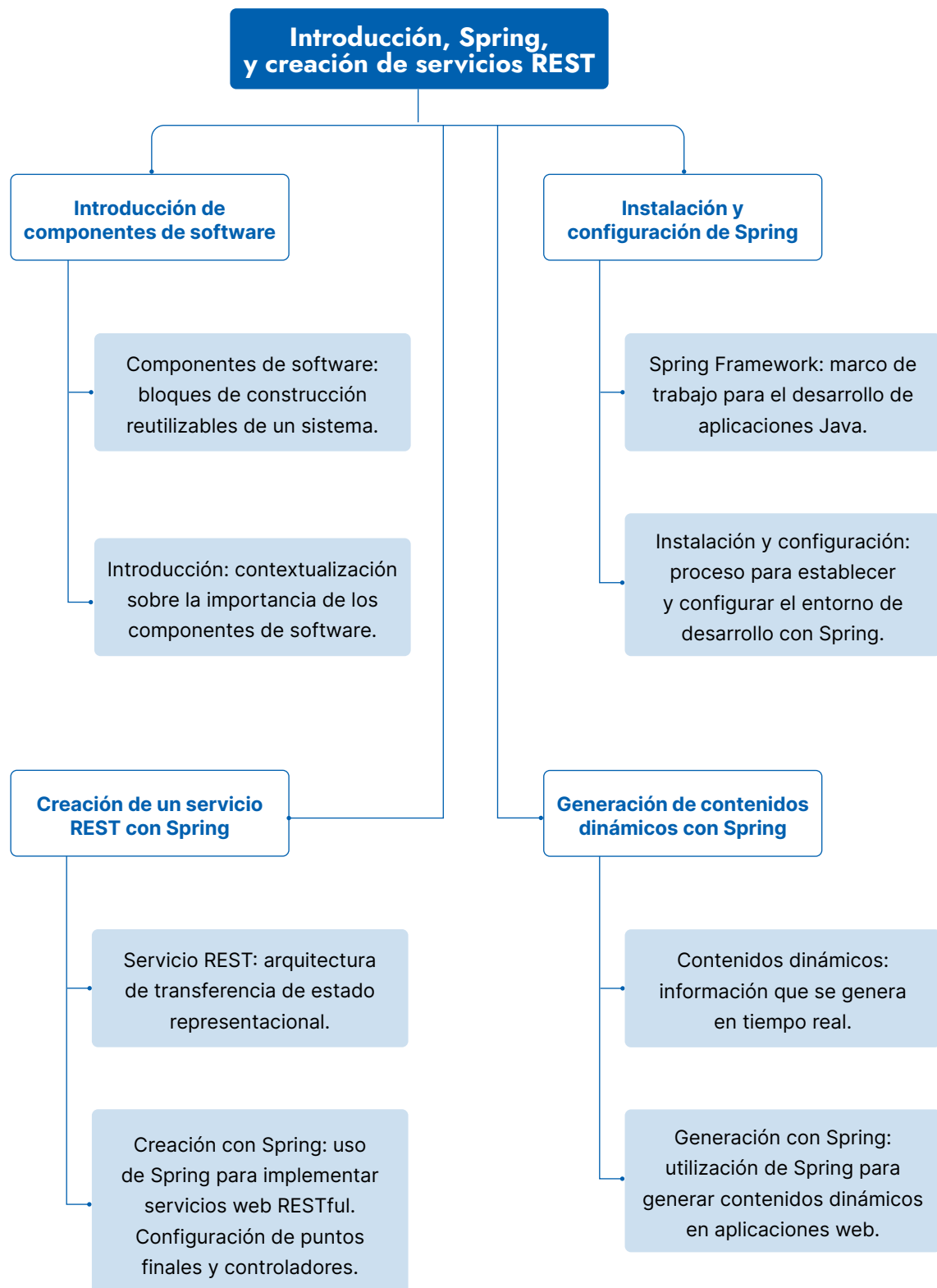
Acceso a datos

ÍNDICE

Mapa conceptual.....	04
1. Introducción de componentes de software.....	05
1.1. Componentes de software	05
1.2. Aplicaciones web.....	06
1.3. Modelo vista controlador	08
A. Modelo REST.....	09
1.4. Spring Boot	11
2. Instalación y configuración de Spring.....	15
2.1. Crear un proyecto. Estructura	15
2.2. El esquema MVC en Spring.....	22
2.3. Recoger parámetros de las peticiones	24
3. Creación de un servicio REST.....	26
3.1. Creación del servicio REST.....	26
A. Configuración del proyecto	26
B. Modelo.....	27
C. Repositorio.....	27
D. Servicio.....	28
E. Controlador	29
3.2. Consulta del servicio con Postman.....	31
4. Generación de contenidos dinámicos con Spring	32
4.1. Creación de una aplicación web con Spring y MVC.....	32
4.2. Estructura del proyecto.....	32
A. La vista	33
B. El motor de plantillas.....	34

C. Navegación	34
4.3. Cambios en el controlador	34
4.4. La vista	36

Mapa conceptual



01 Introducción de componentes de software

1.1. Componentes de software

Cuando comienzas a desarrollar un proyecto, debes realizar un **análisis del problema** que hay que resolver. Entran en juego todos los conceptos que has estudiado sobre análisis de requerimientos, modelado de datos, etc. Todo ello, independientemente de los lenguajes que vas a utilizar.

Muchas veces nos centramos en la resolución del problema, y perdemos de vista detalles del mantenimiento de la aplicación que desarrollaremos, su escalabilidad y la integración con otras aplicaciones y sistemas. Por lo tanto, existe «algo más» por encima de la programación orientada a objetos, el modelo relacional o el paradigma que utilicemos.

IMPORTANTE

La **programación orientada a componentes** está enfocada a fusionar aplicaciones distribuidas, lo que permite la reutilización de componentes, principalmente de acceso a datos.

DEFINICIÓN

Un **componente (SW o HW)** se refiere a una parte de la solución que queda detallada por lo que hace y por cómo se comunica con su alrededor, es decir, su **comportamiento** y su **interfaz**.

Desde el principio de la ingeniería del software, se persigue trabajar y programar con esa perspectiva, reutilizando y adaptando componentes que sabemos que sirven en otros proyectos. Eso implica poder desarrollar estos componentes por separado, de manera que después puedan «ensamblarse» con otros sin perder funcionalidad. Esto nos permitirá el desarrollo de software de manera más rápida y fiable.

Son **características de un componente**, entre otras:

- » **Encapsulado e independencia:** para realizar su tarea no necesita de otro componente, y solo se comunica a través de su interfaz.

» **Independencia:** de software, hardware y plataforma.

» **Reutilizable:** puede utilizarse en distintas aplicaciones.



¿SABÍAS QUE...?

Los compontes se crearon inicialmente para agilizar el desarrollo de aplicaciones gráficas, gracias a Microsoft, con sus modelos OLE y COM, que evolucionaron hacia las componentes OCX, las cuales desembocaron en el actual .NET.

IMPORTANTE

Por su parte, en el mundo Java destacan las **JavaBeans** para la **Java Standard Edition** y los **EJB (Enterprise Java Beans)** para la **Java Enterprise Edition**. Estas son clases que cumplen ciertos requisitos formales, como **getters, setters, datos simples**, etc.

1.2. Aplicaciones web

Hoy en día, las aplicaciones no solo se ciñen a los entornos de escritorio. Móviles, tabletas y dispositivos **lot (Internet of the things)** han propiciado desarrollos para cada plataforma, lo que dificulta a veces el despliegue de la misma aplicación para cada plataforma.

DEFINICIÓN

Las **aplicaciones web** son herramientas que permiten el acceso a servidores web (y, por ende, a los datos) a través de internet mediante una aplicación cliente. Habitualmente, ese cliente será el **navegador** donde se ejecutan dichas aplicaciones.

Aunque estamos situados en un ciclo de desarrollo de aplicaciones multiplataforma, el hecho de que desarrollemos una aplicación web, o parte de ella, no implica que vayamos a desarrollar contenido web puramente dicho. En esta unidad nos vamos a centrar en el acceso y retorno de los datos a aplicaciones que los soliciten, ya sean móviles, clientes, navegadores o de cualquier índole.

La **estructura de una aplicación web** se divide en dos partes:

Frontend

Es la parte del desarrollo que está visible al usuario, y que gestiona la interfaz gráfica de usuario (GUI). El especialista **frontend** se encarga del diseño gráfico de la interfaz, de la interacción con el usuario y de mejorar la UX (**user experience**). En aplicaciones web, utilizaríamos HTML, CSS, Angular y JavaScript embebido en el propio navegador. Para aplicaciones web híbridas, con una aplicación de escritorio, la tarea de desarrollo de la interfaz gráfica podría ser mediante **Python** o **JavaFx**, o incluso mediante tecnologías móviles para Android e iOS.

Backend

Es la parte de desarrollo oculta al usuario. Habitualmente se ejecuta en un servidor web, y está en comunicación con un servidor de la base de datos. El **backend** se encarga de recibir las peticiones desde el **frontend**, y facilitar los resultados solicitados. Existen multitud de tecnologías aplicables para este desarrollo: lenguajes como **Java**, **NodeJS** y **Rust** permiten el desarrollo de estos servidores.

IMPORTANTE

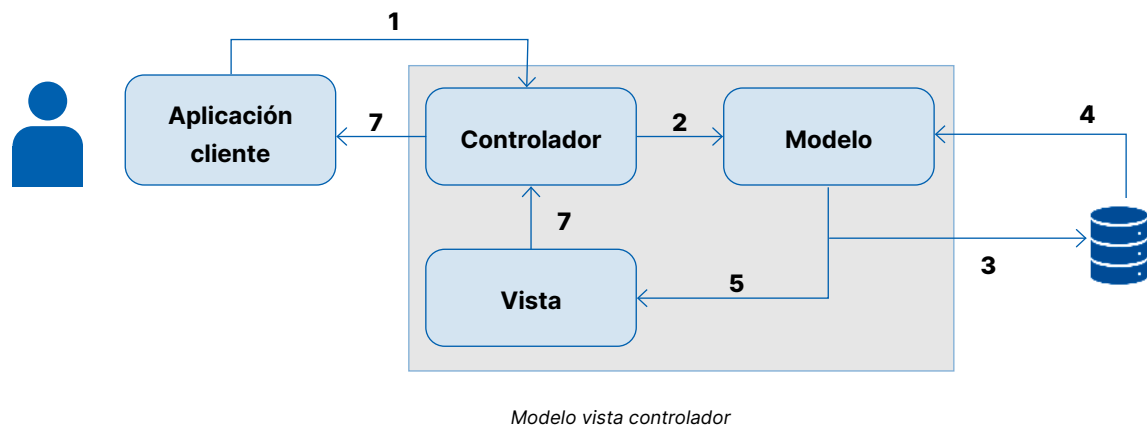
Fíjate en el paralelismo de este modelo con la implementación de servicios, que habrás estudiado en el módulo de programación de servicios y procesos:

- » El usuario interacciona con el cliente y se comunica con el servidor mediante **sockets**.
- » El servidor recoge la petición desde el **socket**, la procesa y le da una respuesta.
- » Ambos deben ponerse de acuerdo en el protocolo y el formato de intercambio de mensajes.

Además de **backend** y **frontend**, aparece el **desarrollo fullstack**, el desarrollo por completo ambas partes. Este concepto es complejo, debido a la rápida evolución de las tecnologías, lo que hace muy complicado conocer completamente las dos visiones de la aplicación. Aun así, debemos **entender ambas partes del diseño** y de la estructura, en vez de ver la otra parte como una caja negra y no preocuparnos de lo que ocurre dentro.

1.3. Modelo vista controlador

El **modelo vista controlador (MVC)** es una arquitectura para el desarrollo de aplicaciones, que separa en capas los datos, la interfaz y la lógica de los controles.



1. El usuario, al utilizar la aplicación cliente (habitualmente un navegador), realiza una **petición** a través del protocolo http (HTTP_REQUEST). Estas peticiones las recibe el módulo del controlador. El controlador gestiona un listado de operaciones o eventos que puede procesar, determinado por el propio protocolo de este.
2. En caso de ser alguna petición que solicita alguna **consulta o modificación de los datos**, pasamos a la capa del modelo los parámetros proporcionados en la petición. Esta capa accede a los datos, realizando los pasos 3 y 4 mediante operaciones al SGBD. El modelo está ahora en disposición de dar una respuesta.
3. **Consulta** a la base de datos.
4. **Recuperación** de la consulta a la base de datos.
5. El modelo transfiere a la vista el conjunto de **resultados obtenidos**.
6. La vista es la responsable de recibir los datos y **presentarlos o formatearlos** de manera adecuada.
7. El controlador devuelve la presentación generada por la vista (que habitualmente será un HTML dinámico) a la aplicación cliente, mediante la petición **HTTP_RESPONSE**.

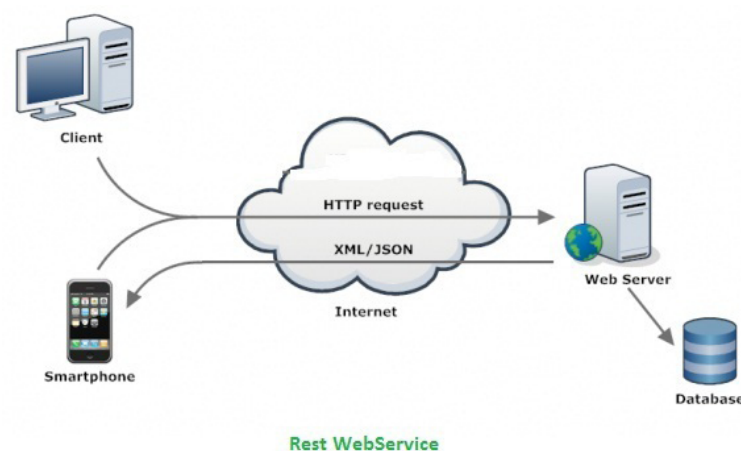
A. Modelo REST

El modelo **REST** (***representational state transfer***) nos permite dotar a nuestro servidor de un servicio para recuperar y manipular los datos de manera sencilla. En dicho modelo, se delega la parte de la vista al cliente, y quedan en el servidor el controlador y el modelo.

En dicho modelo, destacaremos:

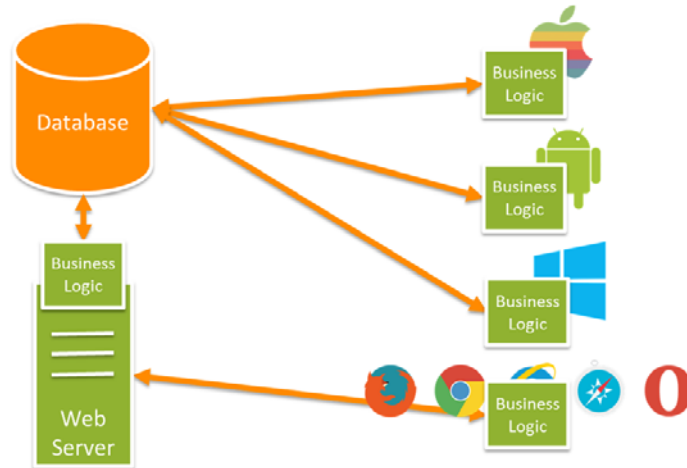
- » **El protocolo sigue el modelo cliente/servidor sin estado (al igual que HTTP):** una petición solo contestará según la información recibida en la misma petición.
- » **Soporte de operaciones CRUD, mediante las especificaciones HTTP equivalentes:** **GET** (consultar), **POST** (crear), **PUT** (modificar) y **DELETE** (borrar).
- » Disposición del modelo **HATEOAS** (***hypermedia as the engine of application state, hypermedia como motor del estado de la aplicación***). Este principio permite incluir en las respuestas los hiperenlaces a los recursos.

Una Web API es una API (Application Programming Interface) implementada para un Servicio Web de forma que éste puede ser accesible mediante el protocolo HTTP, en principio por cualquier cliente web (navegador) aunque existen librerías que permiten que cualquier tipo de aplicación (escritorio, web, móvil, otros servicios web, . . .) accedan a la misma para comunicarse con dicho servicio web.

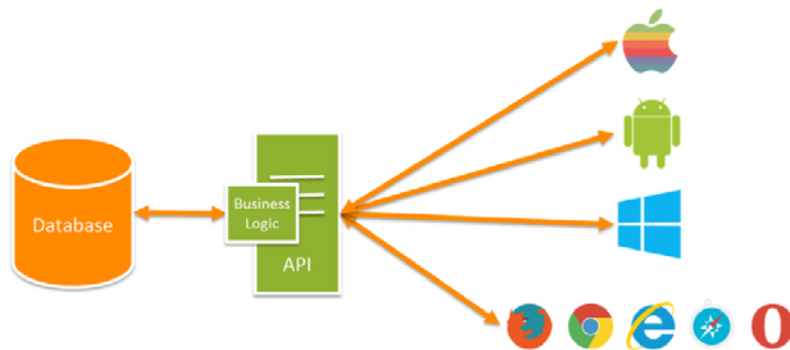


La Web API es una de las partes de los Servicios Web que, tal y como comentábamos anteriormente, mejoran sustancialmente la interoperabilidad de éstos con los potenciales clientes ya que permiten que sólo haya que implementar un único punto de entrada para

comunicarse con el servicio web independientemente del tipo de aplicación que lo haga. De esa manera el desarrollador del Servicio Web define la lógica de negocio en el lado servidor y los diferentes clientes que quieran comunicarse con el mismo lo hacen a través de la Web API realizando solicitudes a las diferentes URLs que definen las operaciones disponibles.



Arquitectura de aplicación web sin API



Arquitectura de aplicación web con API

En comparación con el MVC, sería como eliminar la capa de la vista y devolver los datos procesados por el modelo. Habitualmente, los servidores REST devuelven los datos en formato JSON.

Ejemplo de respuesta a la petición **<https://swapi.dev/api/people/1>** (API pública de Star Wars).

Como puedes observar, al incluir enlaces en la propia API REST se cumple del principio de HATEOAS, para acceder a objetos a partir de la propia respuesta.

Modelo de Capas

El modelo REST (representational state transfer) nos permite dotar a nuestro servidor de un servicio para recuperar y manipular los datos de manera sencilla. En dicho modelo, se delega la parte de la vista al cliente, y quedan en el servidor el controlador y el modelo.

En dicho modelo, destacaremos:

- » **Protocolo cliente/servidor sin estado:** El protocolo sigue el modelo cliente/servidor sin estado (al igual que HTTP): una petición solo contestará según la información recibida en la misma petición.
- » **Soporte de operaciones CRUD:** Se utiliza mediante las especificaciones HTTP equivalentes: GET (consultar), POST (crear), PUT (modificar) y DELETE (borrar).
- » **Disposición del modelo HATEOAS:** Este principio permite incluir en las respuestas los hiperenlaces a los recursos, siguiendo el concepto de “hypermedia as the engine of application state” (hypermedia como motor del estado de la aplicación).

En comparación con el MVC, sería como eliminar la capa de la vista y devolver los datos procesados por el modelo. Habitualmente, los servidores REST devuelven los datos en formato JSON. Además, el modelo REST puede entenderse en términos de capas:

- » **Capa de presentación (o vista):** En el modelo REST, esta capa se delega al cliente. El servidor solo proporciona los datos y no se involucra en cómo se presentan.
- » **Capa de lógica de negocio (o controlador):** En el servidor, esta capa es responsable de procesar las solicitudes del cliente y aplicar la lógica necesaria para interactuar con el modelo de datos.
- » **Capa de datos (o modelo):** Esta capa maneja el almacenamiento y la recuperación de los datos. En un servidor REST, los datos suelen ser almacenados en una base de datos y son accedidos a través de la API REST.

Ejemplo de respuesta a la petición <https://swapi.dev/api/people/1> (API pública de Star Wars). Como puedes observar, al incluir enlaces en la propia API REST se cumple del principio de HATEOAS, para acceder a objetos a partir de la propia respuesta.

1.4. Spring Boot

Para concluir, veremos algunas características del entorno de trabajo escogido para la presente unidad, ya que nos cubrirá todas las partes analizadas.

El concepto de **framework** (banco de trabajo) es el conjunto de herramientas, reglas y convenciones para aumentar la velocidad de desarrollo, automatizando la velocidad de desarrollo, mediante la automatización de tareas. No hay que confundir **framework** con IDE, aunque en ciertas ocasiones los IDE se adaptan a los **frameworks** mediante ciertos **plugins**.

Spring inicialmente simplifica el desarrollo de aplicaciones Java, incorporando mecanismos como estos:

- » **Inyección de dependencias mediante la inversión de control:** básicamente, el programador delega al **framework** la tarea de creación de objetos en sus clases, y es este último el encargado de cargar los objetos e inyectarlos a las clases que los necesitan cuando son necesarios.
- » **Gestión sencilla de componentes POJO y BEANS.**
- » **Programación orientada a aspectos:** separa las diferentes tareas que debe realizar una clase en nuestra aplicación.
- » **Adición de comportamientos a las clases mediante el uso de proxies:** el programador creará objetos básicos, pero, mediante el preproceso de estos proxies, añadiremos comportamientos (métodos) comunes a nuestros objetos.

Conceptos y proporcionar un ejemplo de creación de un BEAN (también conocido como POJO) y su posterior uso.

- » **POJO (Plain Old Java Object):** Un POJO es una clase simple en Java que no implementa ninguna interfaz específica ni extiende ninguna clase especial. Es una clase que solo contiene campos (variables miembro) y métodos getter y setter para acceder y modificar esos campos.
- » **BEAN (JavaBean):** Un JavaBean es un tipo especial de POJO que sigue convenciones específicas, como tener un constructor sin argumentos, proporcionar métodos getter y setter para acceder a los campos, ser serializable, etc. Los JavaBeans son comúnmente utilizados para representar datos en aplicaciones Java y son compatibles con muchas herramientas y frameworks.

BEAN simple en Java:

```
public class Persona {  
    private String nombre;  
    private int edad;
```

```
// Constructor vacío (obligatorio para ser un JavaBean)
public Persona() {
}

// Constructor con parámetros
public Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
}

// Métodos getter y setter para el campo 'nombre'
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

// Métodos getter y setter para el campo 'edad'
public int getEdad() {
    return edad;
}

public void setEdad(int edad) {
    this.edad = edad;
}
}
```

BEAN en una aplicación Java:

```
public class Main {
    public static void main(String[] args) {
        // Crear una instancia de Persona
        Persona persona = new Persona("Juan", 30);
        // Acceder a los datos usando los métodos getter
        System.out.println("Nombre: " + persona.getNombre());
        System.out.println("Edad: " + persona.getEdad());

        // Modificar los datos usando los métodos setter
        persona.setNombre("María");
        persona.setEdad(25);
    }
}
```

```
// Acceder a los datos actualizados
System.out.println("Nombre actualizado: " + persona.getNombre());
System.out.println("Edad actualizada: " + persona.getEdad());
}
}
```

Este ejemplo muestra cómo crear un BEAN llamado **Persona** con dos campos (**nombre** y **edad**) y cómo utilizar los métodos getter y setter para acceder y modificar los datos de un objeto **Persona**.



¿SABÍAS QUE...?

Spring Boot apareció años después, con el objetivo de completar o mejorar a Spring, ya que Spring **framework** sigue funcionando de manera subyacente a Spring Boot. Spring Boot es un acelerador para la creación de proyectos, aplicando el patrón de **convención antes de la configuración**. Este patrón reduce el número de acciones que tome el desarrollador en el momento de la creación del proyecto. El programador decide sobre el comportamiento de su futura aplicación y Spring Boot seleccionará automáticamente las dependencias que necesita. Si se requiere alguna más, dota de flexibilidad al programador para añadir o configurar, pero el objetivo es minimizar el tiempo en esta parte.

El **flujo de trabajo en el desarrollo de aplicaciones web** es:

- 1** | **Seleccionar comportamientos** (y para ello seleccionar todas las dependencias para conseguir dicho comportamiento).
- 2** | **Desarrollar** la aplicación.
- 3** | **Desplegarla** en un servidor.

Spring Boot automatiza prácticamente las fases 1 y 3, ya que incluye un servidor **Tomcat** embebido, y, al lanzar la aplicación, pondrá en marcha dicho servidor. Esto facilita mucho el trabajo y el despliegue mediante contenedores (Docker o similares) en servicios almacenados en la nube.

02 Instalación y configuración de Spring

Spring es un **framework** para el desarrollo de aplicaciones, y **Spring Boot** permite acelerar aún más el proceso de preparación y despliegue de los proyectos Spring.

Es decir, podemos añadir **plugins** a nuestros editores favoritos, como Eclipse, Netbeans o Visual Studio Code, pero en nuestro caso vamos a utilizar **Spring Tool Suite**.



Spring Tools Suite es básicamente una distribución de Eclipse, preparada exclusivamente para el trabajo con Spring, lo que nos evita las tareas de añadir versiones, configurar y administrar los **plugins** (complementos) necesarios. Podemos descargarla desde la página oficial. Esta versión de Eclipse se instala como un **bundle** (empaquetado o autocontenido), incorporando en la misma carpeta donde descomprimos la aplicación tanto los ficheros ejecutables como complementos y herramientas adicionales. Existen versiones gratuitas para Linux, Mac y Windows.

2.1. Crear un proyecto. Estructura

Una vez que hemos instalado la aplicación, y que arrancamos el programa y configuramos la carpeta de trabajo (donde se crearán nuestros proyectos), debemos empezar con la creación de este. Aquí es donde aparece la potencia de **Spring Boot**, ya que no crearemos un proyecto y tendremos que añadir las dependencias que necesitemos, sino que, en el asistente mismo, le indicaremos cómo deseamos que se comporte nuestra aplicación, y el asistente añadirá las dependencias necesarias para su desarrollo.



CLAVES Y CONSEJOS

Por supuesto, podremos añadir nuevas dependencias, aunque no hayamos especificado algún comportamiento, ya que Spring gestiona sus proyectos mediante Maven de manera subyacente. No tendremos más que acudir al fichero POM.XML y añadirla sin ningún problema.

En el momento de la creación, debemos especificar las características del proyecto y añadir los comportamientos deseados. En esta unidad, trabajaremos con **Web** y **DevTool**.

Durante el asistente, se nos pregunta por estos apartados, necesarios al ser un proyecto Maven. Estas opciones serán usadas para referenciar el proyecto de manera completa:

- » **groupID**: viene a ser la organización que desarrolla.
- » **artifactID**: el contenido que estamos desarrollando.
- » **versión**: la versión de este.

Una vez creado el proyecto, Spring realizará las siguientes tareas:

- » Generará el fichero **pom.xml**, con las características indicadas en el asistente.
- » Descargará las dependencias.
- » Creará la jerarquía de las carpetas necesarias para trabajar.
- » Añadirá los comandos de compilación embebidos dentro del **mvnw (maven wrapper)**.

Con esto, nos quedará una **estructura mínima** como sigue:

- » **/src**: contiene el código completo de la aplicación.
- » **main/java**: contendrá el código Java de nuestro proyecto. Aquí ya veremos que lo separaremos en distintos paquetes según la funcionalidad de las clases (modelo, controladores, etc.).
- » **main/Resources**: contendrá el código estático de nuestra aplicación, como pueden ser páginas HTML, estilos, etc., así como ficheros de propiedades y de configuración.
- » **/target**: contendrá nuestro proyecto una vez empaquetado para distribuir.


```
/proyecto-spring
└─ src
    └─ main
        ├── java
        │   └─ com
        │       └─ ejemplo
        │           └─ proyecto
        │               ├── controladores
        │               │   └─ Controlador.java
        │               ├── modelos
        │               │   └─ Modelo.java
        │               └─ servicios
        │                   └─ Servicio.java
        └─ resources
            ├── static
            │   ├── css
            │   │   └─ styles.css
            │   └─ js
            │       └─ script.js
            ├── templates
            │   └─ index.html
            └─ application.properties

/target
```

Explicación de la estructura:

» **/src/main/java**: Contiene el código Java de nuestro proyecto. Aquí se dividen los paquetes según la funcionalidad de las clases. En este ejemplo, tenemos paquetes para controladores, modelos y servicios.

» **com.ejemplo.proyecto.controladores**: Aquí estarán los controladores de nuestro proyecto.

- **com.ejemplo.proyecto.modelos:** Aquí estarán las clases que representan los modelos de datos.
 - **com.ejemplo.proyecto.servicios:** Aquí estarán las clases que contienen la lógica de negocio de nuestro proyecto.
- » **/src/main/resources:** Contiene el código estático de nuestra aplicación, como archivos HTML, CSS, JS, así como archivos de configuración.
- **static:** Contiene archivos estáticos como CSS y JS.
 - **css:** Archivos CSS para estilos.
 - **js:** Archivos JS para scripts.
 - **templates:** Contiene archivos HTML para las vistas de nuestra aplicación.
 - **application.properties:** Archivo de configuración de Spring Boot.
- » **/target:** Contendrá nuestro proyecto una vez empaquetado para distribuir. Este directorio se genera automáticamente cuando compilamos y empaquetamos nuestro proyecto.

Explicación general de la importancia de cada carpeta en la estructura de un proyecto Spring con el modelo API REST:

1. /src/main/java:

- » **controladores (controllers):** Aquí se encuentran las clases que actúan como puntos de entrada para las solicitudes HTTP. Estos controladores manejan las peticiones REST y dirigen el flujo de ejecución a través de la aplicación. Son responsables de recibir las solicitudes, procesarlas y enviar las respuestas adecuadas.
- » **modelos (models):** En esta carpeta se encuentran las clases que representan los modelos de datos de la aplicación. Estas clases definen la estructura de los datos que se manejan en la aplicación y suelen estar relacionadas con las entidades del dominio de la aplicación.
- » **servicios (services):** Aquí residen las clases que contienen la lógica de negocio de la aplicación. Estos servicios encapsulan la funcionalidad de la aplicación y son responsables de realizar operaciones más complejas, como la validación de datos, el procesamiento de información y la interacción con la capa de acceso a datos.

2. /src/main/resources:

- » **static:** Contiene archivos estáticos como CSS, JavaScript, imágenes, etc. Estos recursos se utilizan para la interfaz de usuario de la aplicación, como estilos de diseño, scripts de cliente y otros archivos estáticos.

»**templates:** Aquí se almacenan las plantillas HTML utilizadas para generar las vistas de la aplicación. Estas plantillas pueden contener marcadores o expresiones que serán reemplazadas por datos dinámicos cuando se generen las páginas web.

»**application.properties:** Archivo de configuración de Spring Boot donde se definen propiedades específicas de la aplicación, como la configuración del servidor, la configuración de la base de datos, entre otras.

3. /target: Contiene el código compilado y empaquetado de la aplicación listo para su distribución. Este directorio se genera automáticamente durante el proceso de compilación y empaquetado del proyecto. Es importante para la distribución y despliegue de la aplicación en un entorno de producción.

EL PROGRAMA PRINCIPAL

El asistente nos habrá creado un programa principal, que tendrá el siguiente aspecto:

```
package miPaquete; // Paquete donde se encuentra la clase

// Importaciones necesarias de Spring Boot
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
SpringBootApplication;

// Anotación para indicar que esta clase es una aplicación Spring
Boot
@SpringBootApplication

public class Application {

    // Método principal que arranca la aplicación Spring Boot
    public static void main(String[] args) {

        // Ejecuta la aplicación Spring Boot, pasando la clase
principal (Application) y los argumentos de línea de comandos
        SpringApplication.run(Application.class, args);

    }

}
```

Explicación:

» `package miPaquete;` Declaración del paquete donde se encuentra la clase `Application`. Los paquetes ayudan a organizar y estructurar el código de Java.

» `import org.springframework.boot.SpringApplication;` Importación de la clase `SpringApplication` del paquete `org.springframework.boot`, que es necesaria para arrancar la aplicación Spring Boot.

» `import org.springframework.boot.autoconfigure.SpringBootApplication;` Importación de la anotación `SpringBootApplication` del paquete `org.springframework.boot.autoconfigure`, que es una anotación compuesta que combina varias anotaciones Spring para configurar automáticamente la aplicación.

» `@SpringBootApplication`: Anotación que indica que esta clase es una aplicación Spring Boot. Esta anotación combina las anotaciones `@Configuration`, `@EnableAutoConfiguration` y `@ComponentScan`, lo que permite que Spring Boot configure automáticamente la aplicación y escanee los componentes necesarios.

» `public class Application { ... }` Declaración de la clase `Application`, que es la clase principal de la aplicación.

» `public static void main(String[] args) { ... }` Método principal que se ejecuta cuando se inicia la aplicación. Este método arranca la aplicación Spring Boot llamando al método estático `run` de `SpringApplication`.

» `SpringApplication.run(Application.class, args);` Llama al método estático `run` de `SpringApplication`, pasando la clase principal (`Application.class`) y los argumentos de línea de comandos (`args`). Este método arranca la aplicación Spring Boot.

Tabla de descripción del ciclo de vida de Spring Boot

PASO	DESCRIPCIÓN
<code>SpringApplication.run(Application.class, args)</code>	Método estático que inicia la aplicación Spring Boot. Recibe la clase principal de la aplicación y los argumentos de la línea de comandos como parámetros.
Configuración automática de Spring Boot	Spring Boot realiza una configuración inicial automática basada en las dependencias y convenciones detectadas en el proyecto. Esto simplifica la configuración de la aplicación y reduce la cantidad de código boilerplate.

Carga de configuraciones personalizadas	Spring Boot carga las propiedades de aplicación y las configuraciones específicas definidas en el proyecto, como archivos <code>application.properties</code> , <code>application.yml</code> o <code>bootstrap.properties</code> .
Creación del contexto de aplicación	Spring Boot crea el contexto de la aplicación, que es el contenedor principal de Spring. El contexto de la aplicación gestiona el ciclo de vida de los beans, proporciona acceso a servicios y recursos de la aplicación, y facilita la resolución de dependencias.
Inicialización de beans y servicios	Spring Boot inicializa y configura los beans y servicios definidos en la aplicación. Los beans son objetos gestionados por Spring que representan componentes de la aplicación, mientras que los servicios son beans que proporcionan funcionalidades específicas.
Arranque del servidor web integrado (Tomcat)	Spring Boot inicia el servidor web integrado, que por defecto suele ser Tomcat. El servidor web escucha las solicitudes HTTP entrantes y las dirige a los controladores adecuados de la aplicación.
Aplicación lista y en ejecución	Una vez finalizados los pasos anteriores, la aplicación Spring Boot está lista y en ejecución, preparada para procesar solicitudes de usuarios y realizar las tareas previstas.

Notas adicionales:

- » El ciclo de vida de Spring Boot es un proceso complejo que involucra múltiples pasos y componentes internos. La tabla anterior proporciona una descripción simplificada de los pasos principales.
- » La configuración automática de Spring Boot puede ser personalizada utilizando anotaciones, archivos de configuración o propiedades del sistema.
- » Spring Boot ofrece una amplia gama de funcionalidades y extensiones para facilitar el desarrollo de aplicaciones web robustas y escalables.

Recursos adicionales:

- » Documentación oficial de Spring Boot: <https://start.spring.io/>
- » Tutorial de Spring Boot: <https://www.baeldung.com/spring-boot>
- » Guía de referencia de Spring Boot: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>

La magia de Spring la consigue mediante la anotación `@SpringBootApplication`, que engloba tres anotaciones anteriores:

- » `@Configuration`: define clases de configuración, mediante anotaciones. Permitirá una inyección de dependencias.
- » `@EnableAutoConfiguration`: configurará la aplicación en función de las dependencias añadidas; por ejemplo, si encuentra una dependencia de un SGBD y no se ha configurado ningún **bean** de conexión a datos, creará uno automáticamente.
- » `@ComponentScan`: buscará componentes dentro del paquete que tenemos anotado como `@Configuration`.

Al ejecutar el programa, hemos de fijarnos en la salida por consola y en los siguientes detalles:

```

  ____ _
 / ___ \| | | |
/ /___ \| |_| |
\___)___|_____|
:: Spring Boot :: (v2.6.3)

... o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080
(http) with context path ''
```

Spring ha arrancado un servidor Tomcat, donde ejecutará su servicio. Dicho servidor está escuchando en el puerto **8080** por defecto.

- » **Documentación oficial de Tomcat:** <https://tomcat.apache.org/tomcat-8.5-doc/index.html>
- » **Tutorial de Tomcat para principiantes:** <https://www.tutorialspoint.com/apache-tomcat-application-administration/index.asp>
- » **Guía de instalación de Tomcat:** <https://tomcat.apache.org/>
- » **Ejemplos de aplicaciones web Java con Tomcat:** <https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/>
- » **Foro de la comunidad de Tomcat:** <https://tomcat.apache.org/findhelp.html>

2.2. El esquema MVC en Spring

Partimos de que estamos modelando una aplicación web, por lo que esta aplicación estará escuchando en un servidor en un determinado puerto. Por defecto, la dirección a lo largo de la unidad será `http://localhost:8080`. Posteriormente, en caso de realizar el despliegue, revisaremos la URL. Podemos modificar el puerto donde escuchará Tomcat las peticiones, editando el fichero `application.properties`, en la ruta `/src/main/resources`, y añadir la propiedad `server.port=9090` (o el número que deseemos).

A continuación, para dotar al proyecto del patrón MVC (modelo vista controlador), debemos crear las clases pertinentes para modelar las posibles rutas que hay que controlar. Para ello, se creará un paquete con el mismo prefijo que el paquete de la aplicación principal, pero añadiéndole un campo «**.controllers**» (o **.controlador**, si lo deseas en castellano).

El objetivo es tener las funcionalidades del proyecto separadas por paquetes.

Debemos indicar la clase que funcionará como controlador mediante anotaciones, tanto a la clase como a los métodos:

ANOTACIÓN	SIGNIFICADO
@Controller Afecta a Clase	Hereda de @Component , lo que permite que sea detectada explorando el classpath de la aplicación. Contendrá manejadores de eventos.
@RequestMapping("ruta") Afecta a Clase	Define que todos los métodos dentro de esta clase estarán mapeados dentro de una ruta a partir de la URI de la aplicación; por ejemplo, <code>http://localhost/ruta/</code> .
@XXXMapping("subRuta") Afecta a Método	Indica que dicho método mapeará la petición http-XXX (donde XXX es Get , Post , Put o Delete). Dentro del cuerpo de dicho método, programaremos la lógica de cómo tratar dicha petición.
@ResponseBody Afecta a Método	Indicamos al método que el objeto que devolverá se serialice a JSON y se devuelva a quien formalizó la petición. Con la ausencia de esta anotación se devolverá un Modelo con los datos empaquetados para que la vista los renderice.
@RestController Afecta a Clase	Esta anotación se utiliza cuando todos los mapeos de dicho controlador son peticiones REST . Nos permitirá eliminar de los manejadores de peticiones la anotación @ResponseBody .



En versiones antiguas de Spring hay un método genérico donde indicamos, como parámetros, la ruta y el método que debe activarlo: **@RequestMapping(value="ruta", method=RequestMethod.XXX)**, donde **XXX** será **Get**, **Put**, **Post**, **Delete**.

2.3. Recoger parámetros de las peticiones

Visto ya cómo podemos mapear las peticiones HTTP de los usuarios, nos queda ver cómo podemos **recuperar los argumentos** que vendrán con dicha petición. Estos argumentos los definiremos como argumentos del manejador de cada petición como sigue.

Deberemos poner tantos argumentos del manejador como argumentos tendrá la petición HTTP. Veamos los ejemplos:

```
@GetMapping("/ejemplo")
public String manejador(@RequestParam(required=false,defaultValue
= "vacío") String codigo) {
    return "El valor de tu código es " + codigo;
}

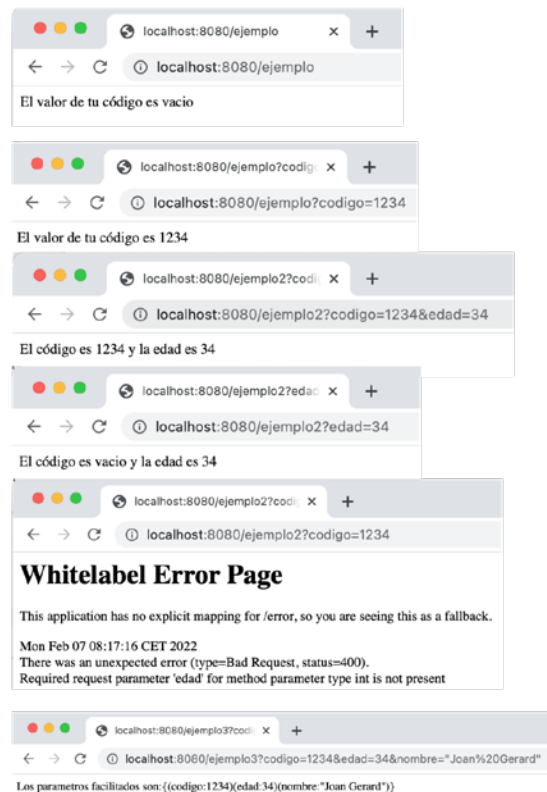
@GetMapping("/ejemplo2")
public String manejador(
    @RequestParam(required=false,defaultValue = "vacío") String
codigo,
    @RequestParam(required=true,name="edad") int age) {
    return "El código es " + codigo + " y la edad es " + age;
}

@GetMapping("/ejemplo3")
public String manejador(@RequestMapping(@RequestParam
Map<String,String> todosEnUno) {...})
```

En el primer caso tenemos un atributo opcional, con valor por defecto "", de nombre id. Las llamadas podrían ser **http://localhost:8080/ejemplo** o bien **http://localhost:8080/ejemplo?codigo=1234**

En el segundo caso tenemos dos, pero el primero es opcional. El segundo parámetro difiere del nombre de la variable, con lo que las llamadas podrían ser: **http://localhost:8080/ejemplo2?edad=34** o **http://localhost:8080/ejemplo2?edad=34&id=codigo**. En los resultados de ejecución en la siguiente imagen, fíjate en qué ocurre cuando no existe el parámetro edad, que hemos definido como obligatorio.

En el tercer caso, recibiremos un mapa de parámetros (formado por parejas clave-valor), a los que tendremos que acceder a través de los métodos de recuperación de este.



CLAVES Y CONSEJOS

Como finalización de lo visto anteriormente, los parámetros que podemos recuperar son los que vienen en la línea de la URL, pero esta práctica se desaconseja, debido a que se ve el valor de dichos parámetros. La alternativa es enviarlos en el cuerpo de la petición (**body**).

Lo que deberíamos hacer es simplemente cambiar `@RequestParam` por `@RequestBody`, sin pérdida de generalidad. Los parámetros que vienen como variables en la URL serán anotados con `@PathVariable`, como se verá en las próximas unidades.

03 Creación de un servicio REST

En este apartado vamos a **crear un servicio REST con Spring Boot**. Para ello, lo primero que tenemos que entender son las componentes que vamos a utilizar y cómo las enlazaremos entre ellas.

Hay que indicar también que la creación de un proyecto de un servicio REST implica muchos detalles y casuísticas que contemplar, que escapan a la unidad, incluso al módulo entero.



Mientras desarrollamos los contenidos del tema, se irán viendo capturas o bloques de código de un proyecto de ejemplo. El proyecto versa sobre la base de datos de Cine, con datos de películas y directores de estas.

3.1. Creación del servicio REST

Un proyecto completo de Spring, cuando se aborda por primera vez, puede resultar apabullante, debido a la cantidad de clases que entran en juego. Por esta razón, vamos a explicar poco a poco cada una de estas clases y su separación en paquetes, así como los objetivos de estas.

A. Configuración del proyecto

Cuando creamos un proyecto con Spring, tenemos que indicar los comportamientos que va a tener.

En el **asistente de creación** (o bien mediante Spring Initializr) deberemos seleccionar los siguientes arquetipos: **DevTools**, **Lombok**, **MySQL** (o el SGBD preferido), **Spring Data JPA** y **Spring Web**.

Con todos ellos, el asistente nos crea la estructura del proyecto y el fichero pom.xml, dado que **Spring** desarrolla por defecto con **Maven**. En principio, este fichero no hará falta modificarlo, aunque sí revisarlo.

A continuación, deberemos configurar la parte de puerto de escucha del servidor, configuración de la base de datos y configuraciones de **Hibernate**.

B. Modelo

Spring permite mapear nuestras bases de datos de manera muy cómoda con **Hibernate** y **anotaciones JPA**, por lo que la parte de acceso a la base de datos se realizará de la misma manera.

Aquí, en el modelo, es donde estarán las clases conocidas como beans (clases con atributos, constructores, **getters** y **setters** y algunos métodos más).

Estas clases podrán estar relacionadas entre ellas con las anotaciones de relaciones **@OneToOne**, **@OneToMany** y **@ManyToMany**, sin pérdida de generalidad.

C. Repositorio

El repositorio es una clase fundamental en el proceso de creación de aplicaciones con acceso a base de datos. **Un repositorio contendrá los métodos que permiten acceder a las operaciones básicas, como cargar, guardar, borrar y actualizar.**

Con Spring nos ahorraremos mucho trabajo, ya que simplemente definiremos una interfaz de trabajo, en la cual indicaremos el tipo de repositorio que queremos crear, la clase sobre la cual va a trabajar y el tipo de datos que funciona como identificador de dicha clase:

```
public interface classRepositorio extends xxxRepository<Class t,
Type t>{
    // dejarlo vacío
}
```

Spring provee de la interfaz **Repository**, de quien hereda **CrudRepository**, que incluye la definición de las operaciones CRUD básicas. De esta última hereda **PagingAndSortingRepository**, que añade funciones de paginación y ordenación, y por último tenemos **JpaRepository**, que incluye operaciones específicas para JPA.

La importancia de la definición genérica del **Repositorio<Clase, Tipo>** es que todos los objetos que recuperará son de dicha clase, y el tipo indica el tipo de la clave principal de dicha clase. Siguiendo nuestro ejemplo, la definición del repositorio sería:

```
public interface DirectorRepo extends JpaRepository<Director, Long>{ }
```

Con esto, Spring ya nos permite acceder a la base de datos y realizar las operaciones básicas. Los métodos siguientes vienen implementados por defecto, y **no necesitaremos implementarlos**, solo definirlos:

Recuperar datos

`findAll()`, `findById(Id)`, `findById(Iterable<Id>)`: recupera una o todas las ocurrencias de un identificador o una colección de identificadores.

Borrar datos

`delete(Object)`, `deleteAll()`, `deleteById(Id)`, `deleteAllById(Iterable<Id>)`: borran según el objeto identificado o todos.

Contar y comprobar

`count()`, `existsById()`

Guardar

`save(Object)`, `save(Iterable<Object>)`: guarda el objeto u objetos.

**CLAVES Y CONSEJOS**

Puedes añadir funciones personalizadas más concretas definiendo un query, que se ejecutará directamente sin tener que programarlo (a menos que tenga argumentos).

D. Servicio

El servicio, o la declaración de una clase mediante el estereotipo `@Service`, indicará que dicha clase gestiona las operaciones de la lógica de negocio. Estos servicios son los que contendrán las llamadas al repositorio.

Para la creación de servicios (y después los controladores), podemos hacerlo de dos maneras:

- » Programando e implementando la clase directamente.
- » Definiendo previamente una interfaz con los métodos que haya que implementar y posteriormente la clase que implemente dicha interfaz.

Esta segunda manera sigue el **patrón FACADE (fachada)**; podrás encontrarla en muchos ejemplos, pero queda fuera del alcance del curso.

```
// fachada del servicio
public interface DirectorService {
```

```
public List<Director>findAllDirector();
public Optional<Director> findDirectorById(Long id);
public Director saveDirector(Director nuevoDirector);
public String deleteDirector(Long id);
public String updateDirector(Director nuevoDirector);
}
//implementación del servicio
@Service
public class DirectorServiceImple implements DirectorService{

    @Autowired
    DirectorRepo directorRepositorio;

    @Override
    public List<Director>findAllDirector() {
        return directorRepositorio.findAll();
    }
    @Override
    public Optional<Director> findDirectorById(Long id) {
        return directorRepositorio.findById(id);
    }
}
```



Como se puede comprobar, el servicio se encarga de invocar los métodos del repositorio, y de realizar algunas comprobaciones en caso de que sea necesario.

E. Controlador

El controlador será el encargado de responder a las peticiones del usuario con la aplicación.

Incluirá los servicios, y en caso de crear una **aplicación MVC** podrá invocar motores de plantillas, como **Thymeleaf**, que estudiaremos en el siguiente apartado. Como hemos comentado aquí, lo implementaremos en una sola clase (sin seguir el patrón anterior).

El controlador invocará al servicio asociado a dicha petición y devolverá los datos obtenidos o la respuesta al mismo cliente. Debemos marcar la clase con el estereotipo **@Controller**.

Para el caso de servicios **REST**, además debemos indicar que las devoluciones de los métodos de la clase sean serializadas a **JSON**, y eso lo conseguimos con **@ResponseBody**.



¿SABÍAS QUE...?

Desde la versión 4 de Spring, las dos anotaciones se han fusionado en una, mediante **@RestController**. Dejaremos solo **@Controller** para proyectos en los que devolvamos una vista (HTML + CSS + JS).

Así pues, el controlador quedaría:

```
@RestController
public class DirectorController {
    @Autowired
    private DirectorService directorService;

    @GetMapping(value = "/director")
    public List<Director> getDirector() {
        return directorService.findAllDirector();
    }

    @GetMapping(value = "/director/{id}")
    public Optional<Director> getDirectorById(@PathVariable Long
id) {
        return directorService.findDirectorById(id);
    }
    // resto de métodos
}
```

IMPORTANTE

Aquí aparece la clase de envoltura **Optional<T>**. Es una clase que, básicamente, contiene un objeto de tipo T. Esta clase tiene dos métodos, **isPresent()** y **get()**, que nos sirven para saber si se ha devuelto algo (no es **null**) y para recuperar el objeto, respectivamente.

Podemos mejorar los métodos para el caso de que no existan resultados o haya ocurrido algún error, encapsulando los resultados en un **ResponseEntity<Resultado>**. Esta clase devuelve el resultado a la aplicación cliente, pero permite añadir un argumento, que será

el código de estado HTTP. Este código puede capturarse en el cliente para la gestión de errores.

```
public ResponseEntity<Result> metodoDelControlador() {  
    // recogemos los datos del servicio/repositorio  
    if (!error) {  
        return new ResponseEntity<>(Resultados, HttpStatus.OK); //  
        TODO BIEN  
    }  
    return new ResponseEntity<>(HttpStatus.NOT_FOUND); // ALGO  
    FALLA }  
}
```

IMPORTANTE

Tanto en el servicio como en el controlador, aparece la anotación `@Autowired`, que se encarga de inyectar el código necesario de aquella clase que implemente el servicio. Mediante esta anotación se crean los objetos necesarios sin necesidad de que lo realice el programador, de manera automática.

3.2. Consulta del servicio con Postman



VÍDEO LAB

Para consultar el siguiente vídeo sobre API Rest con Postman, escanea el código QR o [pulsa aquí](#).

En este apartado vamos a utilizar **Postman** como herramienta de prueba de servidores **REST**, ya que con el navegador solo podemos probar las peticiones **GET**.

04 Generación de contenidos dinámicos con Spring

4.1. Creación de una aplicación web con Spring y MVC

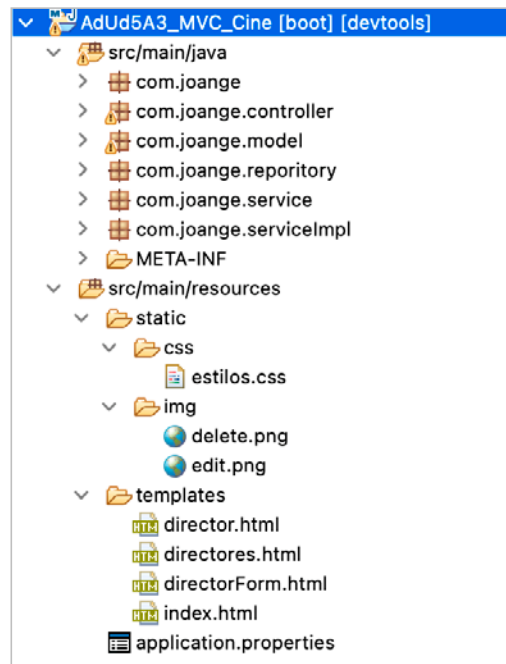
En la unidad anterior hemos construido un API REST con Spring. Como se vio, la aplicación permite gestionar las peticiones de las aplicaciones cliente (realizadas mediante el programa Postman, navegador o similar) y devolver los datos en formato JSON. Estos datos devueltos en crudo deberán ser presentados a los usuarios mediante su interfaz de usuario (GUI) en las aplicaciones cliente.

Lo que vamos a realizar ahora es la propia **presentación de los resultados** (que no de los datos) en una interfaz web, añadiendo la parte de la vista a nuestra aplicación. Además, esa vista se integrará con la vista para realizar también peticiones al controlador y a la lógica de negocio. El resultado será una aplicación web, en la que añadiremos la parte de la vista, y así cubrir la totalidad del patrón MVC.

4.2. Estructura del proyecto

Si recordamos, el flujo de eventos que ocurren en un programa diseñado mediante MVC es el siguiente:

1. El usuario realiza la petición, atendida por el controlador.
2. El controlador decide qué servicio tiene que atender dicha petición, y mediante el modelo recuperará los datos de la base de datos.
3. El modelo devuelve los datos y los envía a la vista que ha indicado el controlador. Este envío de datos del controlador a la vista es el que presenta la gran novedad, dado que los datos se empaquetarán en un objeto especial, denominado **Model**, y se cargará la vista.
4. La vista para generar dicha presentación representará los datos desempaquetando el objeto **Model** recibido, y finalmente será devuelta al cliente.



Así pues, en nuestro proyecto, y partiendo del proyecto anterior, existen tres paquetes que no vamos a modificar para nada, que son el **modelo**, el **repositorio** y el **servicio**. Los datos van a seguir siendo los mismos, y las operaciones de consulta y recuperación desde la base de datos tampoco van a variar. La tarea la tenemos en el controlador y en la vista; sobre todo en esta última, que tenemos que crear desde cero.

A. La vista

Las vistas de nuestra aplicación se encontrarán en la carpeta **src/main/resources**. En esta carpeta hasta ahora teníamos el fichero **application.properties**, pero añadiremos dos subcarpetas:

- » **Recursos estáticos:** son contenidos que no van a variar, como su nombre indica. Nos referimos a imágenes, estilos (y, si fuera necesario, animaciones y sonidos). Dichos contenidos serán cargados, bien por la aplicación, bien por las plantillas.
- » **Plantillas:** estarán en la carpeta templates. Dichas plantillas parecen documentos HTML, pero están programadas para generarlos de manera dinámica, después de procesarlos por el motor de plantillas. En esta unidad vamos a utilizar Thymeleaf como motor de plantillas. Podemos organizarlos también en subcarpetas, para tener organizadas las plantillas de cada apartado.

B. El motor de plantillas

Thymeleaf es un motor de plantillas que, a partir de un código HTML anotado y un modelo de datos, se encarga de generar un HTML completo, que será entregado al cliente como resultado final. Thymeleaf debe añadirse en la creación del proyecto como dependencia de este (con Spring Initializr o con el asistente de Spring Boot Studio).

C. Navegación

Ante el reto de diseñar una página web, hay muchos factores importantes. Uno es la **navegabilidad entre pantallas**.

4.3. Cambios en el controlador

Veamos los cambios necesarios en el controlador. Partiremos, para los ejemplos, del resultado del Apartado 3, donde hemos programado una API Rest que nos gestiona información de directores de cine. El primer cambio que hay que realizar está en la propia definición de la clase controlador:

Rest	MVC
<pre>@RestController public class nombreControlador</pre>	<pre>@Controller public class nombreControlador</pre>

En el API Rest, la anotación **@RestController** equivale a anotar la clase como **@Controller** y además indicar que todos los métodos tienen la anotación **@ResponseBody**. Esta anotación indicaba que el valor devuelto por el método se convierte en JSON y que dicha respuesta se devuelve al cliente que solicitó la petición. Esto solo tiene sentido si trabajamos con **REST**.

Al dejar la anotación con solo **@Controller**, los métodos deberán devolver ahora un **string**, en vez de cualquier tipo de dato. Dicho string será **el nombre de la vista que deberá generarse** y devolverse al cliente. Además, existirá un argumento en los métodos controladores, de tipo **Model**.

Dicho objeto funciona como un **Map** (**atributo - valor**), y deberán añadirse los datos necesarios para generar la vista de manera adecuada por el motor de plantillas. Estos datos se añadirán de manera sencilla con un método `model.addAttribute(etiqueta, valor)`. Dicha etiqueta será el mecanismo de acceso al valor desde la plantilla de la vista.

Aunque tengamos la anotación `@Controller`, esto no significa que no necesitemos implementar algún método que tiene que devolver un objeto JSON al cliente, por ejemplo, para validarse, anotarlo con `@ResponseBody`, y poder convivir métodos que devuelven vistas o JSON.

Supongamos que tenemos una clase `Producto` y que tenemos completamente implementada la parte del servicio, repositorio y modelo. Vemos el ejemplo de la petición de recuperar a todos los productos, comparándola con el controlador **REST**.

```
@GetMapping(value = "/producto")
public List<Producto> getProducto() {
    return productoService.findAllProducto();
}
```

Ante la petición de **GET** `/producto`, se accede al servicio de recuperar todos los productos (`findAllProducto`), que, como vimos en el tema anterior, accede al repositorio y, en consecuencia, a la base de datos. La colección que retorna el servicio se devuelve a la petición. Como dicho método está dentro de la clase `@RestController`, esa lista de productos se transforma a JSON.

```
@GetMapping(value = "/producto")
public String getProducto (Model model) {
    List<Producto> losProductos= productoService.findAllProducto();
    model.addAttribute("productos", lossProductos);
    return "productosView";
}
```

Hemos de fijarnos en el cambio del prototipo de la función, que devuelve un **string** y tiene el **Model** como argumento. Se invoca de forma exactamente igual el servicio. Esto nos demuestra que el Servicio, el Modelo de datos y el Repositorio no han cambiado para nada. Añadimos la lista devuelta al **Model**, con la etiqueta

«**productos**», y devolvemos un **string** «**productosView**». Al **Model** se le puede añadir toda la información que sea necesaria.

Este último valor devuelto dispara el motor de plantillas **Thymeleaf**, que busca en sus recursos un archivo «**productosView.html**» (la extensión no hace falta indicarla). A partir de aquí, Thymeleaf procesará la plantilla con el **Model** recibido, que contiene la colección de productos, y generará el HTML definitivo que será devuelto al cliente.

4.4. La vista

Vamos a estudiar el último eslabón de la cadena, que es la vista. Como hemos comentado, la vista está compuesta por **plantillas Thymeleaf**, que combina HTML con código para procesar los datos recibidos del controlador.

Supongamos, para nuestro ejemplo, que vamos a crear una vista para mostrar una serie de productos (descripción y precio) que hemos recuperado de la base de datos con un controlador (que suponemos implementado). También se facilitará el rol del usuario actual, y, si es un «**admin**», mostraremos el stock de dicho producto. Así pues, el controlador sería algo como:

```
@GetMapping(value = "/productos") public String
getProductos(Model model) {}

List<Producto>losProductos= ...;

model.addAttribute("productos", losProductos);

    model.addAttribute("admin", user.role=="admin");    // true o
false
    return "productos";
}
```

La estructura de la plantilla es un simple HTML, pero debemos indicar que cargue las plantillas de sintaxis de Thymeleaf. En la cabecera podemos añadir todo lo necesario, y en el cuerpo, el contenido.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"><!-- carga de sintaxis
Thymeleaf-->
    <head>    </head>
    <body>
<h1>Listado de productos</h1>
<table>
    <thead>
        <tr>
            <th>Descripcion</th>
            <th>Precio</th>
            <th th:if="${admin}">Disponible</th>
        </tr>
    </thead>
    <tbody>
```

Descripcion	Precio	Disponible
Chair	\$25,00	18-feb-2013
Table	\$150,00	15-feb-2013
Armchair	\$85,00	20-feb-2013
Wardrobe	\$450,00	21-feb-2013
Kitchen table	\$49,00	15-feb-2013
Bookcase	\$80,00	17-feb-2013

```

<tr th:each="producto : ${losProductos}">
    <td th:text="${producto.descripcion}">Descripción</td>
    <td th:text="${producto.precio}">Precio</td>
    <td th:if="${admin}">
        <span th:text="${producto.disponible}">Disponible</span></td>
    </td>
</tr>
</tbody>
</table>
</body>

```

Conceptos claves:

- » La cabecera de la tabla se generará siempre. La tercera columna, solo cuando el `if` se cumple, es decir, cuando **admin** tenga el valor a `true`. También se aplica dentro de los datos.
- » En la creación de las filas de la tabla (`<tr>`) está la etiqueta `th:each`, que realiza un bucle para cada producto de la lista, lo que generará una fila por cada producto. El valor de cada objeto individual es el primer argumento del `each`, mientras que el segundo es la colección de productos.
- » El contenido de una etiqueta se sustituye por el texto del contenido de las variables.

FORMULARIOS CON THYMELEAF

Los datos que se envían a las vistas pueden plasmarse como una página web, para solo visualizarse, o mediante un formulario, de forma que el usuario final puede editar los datos. Para este caso, la sintaxis cambia un poco.

Imaginemos un ejemplo en que se solicitan los datos de un producto; permitimos al usuario que los edite, para posteriormente mandarlos a otra parte del controlador para guardarse:

```

<form th:action="@{/productos/save}" th:object="${producto}"
method="POST">
    <div th:if="*{idProd}"> ID: <input type="text"
th:field="*{idProd}" readonly></div>
    <div> Descripcion: <input type="text"
th:field="*{descripcion}"></div>
    <div> Precio: <input type="text" th:field="*{precio}"></div>
    <input type="submit" th:value="${nuevo}?'Guardar':'Modificar'"/>
</form>

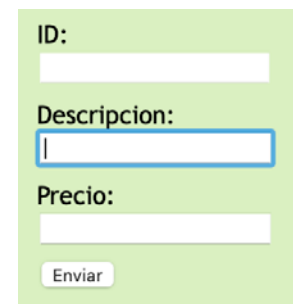
```

Este formulario deberá recibir dos variables: **nuevo**, un valor booleano para indicar si estamos ante un alta o una modificación, y el objeto **Producto**, que pasaremos, o bien con información para modificarlo, o bien con un `new Producto()` para darlo de alta.

» Si el producto es nuevo, hay que tener en cuenta que no poseerá un identificador, por lo que no existirá el cuadro para poner su identificador, ya que se genera automáticamente. Si ya existe, se muestra, pero es **readonly**, ya que el identificador no lo podemos modificar.

» Fíjate en que en el formulario se declara un `th:object="${producto}"`, y todos los campos están identificados por el nombre de sus atributos (`idProd`, `descripcion` y `precio`) mediante `th:field`. Esto es importante para poder **recoger los valores** al enviar los datos.

» El valor de la booleana **nuevo** permite generar la etiqueta el botón de enviar como Guardar/Modificar.



Formulario de producto con los siguientes campos:

- ID:
- Descripción:
- Precio:
- Botón: Enviar

Finalmente, el botón de enviar genera una petición POST a `/productos/save`, que vemos a continuación:

```
@PostMapping(value = "/productos/save") public
String updateDirector(Model model,
    @ModelAttribute("producto") Producto elProducto) {
    productoService.saveProducto(elProducto);
    return "redirect:/productos";
}
```

La novedad está en la anotación `@ModelAttribute("producto") Producto producto`, donde se indica que tenemos un elemento en la plantilla que se llama **producto**, y que corresponde a un objeto de tipo `Producto`, y accedemos a él dentro del método llamándolo **elProducto**. En la función, simplemente guardamos el producto a través del servicio correspondiente y se redirecciona al índice de la página.

UAX FP