
Java Spring MVC

Desarrollo Web en Entorno
Servidor

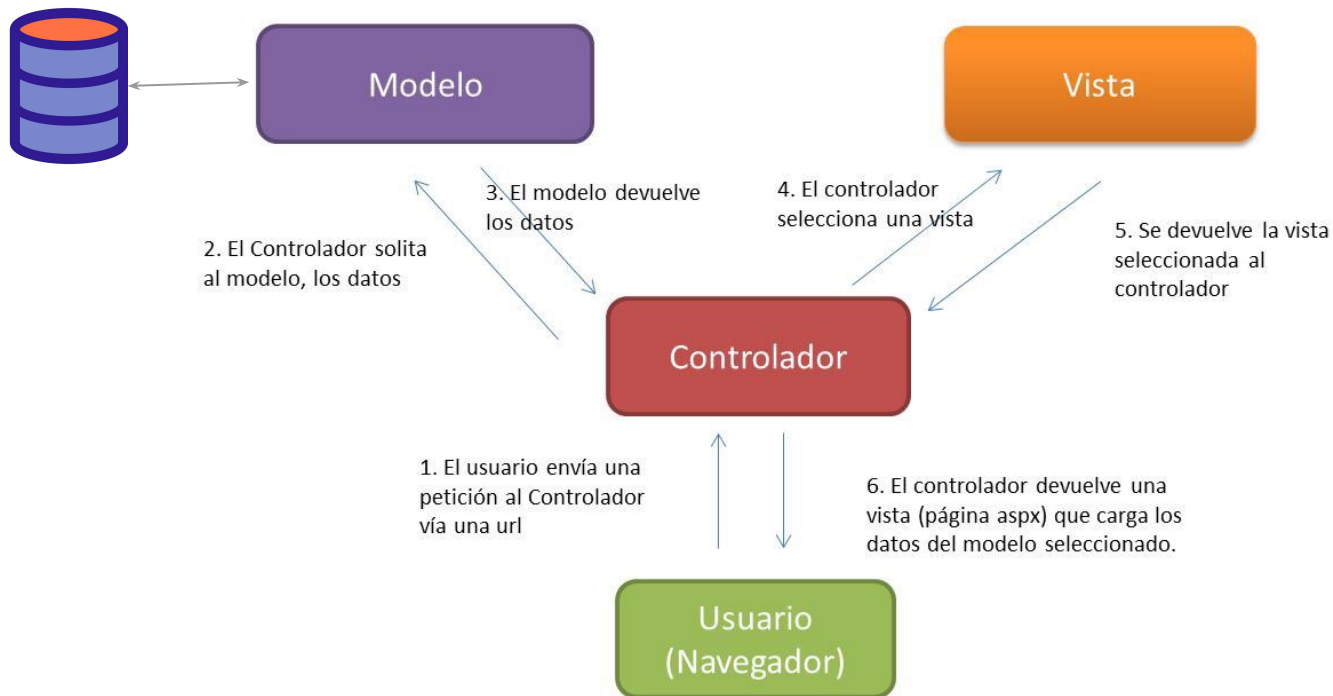
MVC

MVC es un patrón de arquitectura software que separa una aplicación en tres componentes principales: modelos, vistas y controladores.

Esto permite la separación de intereses en cada uno de estos elementos (MVC) y repartir las responsabilidades de una formas más fácil de codificar, depurar y probar, ya que cada una de estas partes tiene solo un trabajo.

En una aplicación MVC, la vista solo muestra información. El controlador controla la entrada y la interacción del usuario y responde a sus peticiones. El modelo se encarga de validar los datos de un modelo e interactuar con la base de datos.

MVC



MVC

Modelos

- Son clases que representan los datos de la aplicación.
- Las clases de modelo usan lógica de validación para aplicar las reglas de negocio para esos datos.
- Normalmente, los objetos de modelo recuperan y almacenan el estado del modelo en una base de datos.
- Los datos actualizados se escriben en una base de datos.
- El modelo actualiza los datos en una base de datos, ya sea para crear, leer, actualizar o eliminar estos datos.
- Estas son las funciones básicas de la persistencia de información en bases de datos, a las que nos referiremos con el acrónimo CRUD (Create-Read-Update-Delete)

MVC

Vistas

- Son los componentes que muestran la interfaz de usuario (IU) de la aplicación.
- Por lo general, esta interfaz de usuario muestra los datos del modelo.
- Utilizaremos el motor de plantillas Thymeleaf para generar las páginas dinámicas con los datos del modelo.

MVC

Controladores

- Son clases que controlan las solicitudes que el usuario realiza a través del navegador.
- Recuperan datos del modelo.
- Llamam a plantillas de vista que devuelven una respuesta.

Terminología Java Utilizada

- Spring
- Spring Core
- Spring Boot
- Spring Initializr
- Thymeleaf
- JDBC
- JPA
- Hibernate

Crear Nuevo Proyecto Spring

Utilizaremos la herramienta **Spring Initializr** para poder generar un proyecto **Spring Boot** de forma rápida.

Podremos utilizar Spring Initializr con la herramienta integrada en nuestro IDE (en eclipse → Spring Tools, o en IntelliJ Idea Community → Spring Initializr) o bien usar la versión web de Spring Initializr (<https://start.spring.io/>) para generar el paquete que descargaremos y abriremos con el IDE.

Al crear un nuevo proyecto Spring con el inicializador especificaremos:

- Tipo de proyecto (Maven, Gradle-Groovy o Gradle-Kotlin)
- Lenguaje (Java, Kotlin o Groovy)
- Versión de Spring Boot (3.2.0 en nuestro caso)
- Java version
- Group, Artifact, Name, Description, Package Name, etc
- Dependencias: En función de las necesidades del proyecto

Crear Nuevo Proyecto Spring Web MVC

Para crear un proyecto web que soporte vistas, incluiremos las dependencias:

- Spring Web
- Thymeleaf

Generamos el
archivo .zip que
descomprimos y
abrimos con IntelliJ
Idea

The screenshot shows the Spring Initializr web application interface. The URL in the browser is <https://start.spring.io>. The interface is divided into several sections:

- Project:** Includes radio buttons for `Gradle - Groovy`, `Gradle - Kotlin`, and `Maven` (which is selected).
- Language:** Includes radio buttons for `Java` (selected), `Kotlin`, and `Groovy`.
- Spring Boot:** Includes radio buttons for `3.0.2 (SNAPSHOT)`, `3.0.1` (selected), and `2.7.8 (SNAPSHOT)`.
- Project Metadata:** Includes input fields for `Group` (filled with `es.cifpcm`), `Artifact` (filled with `HolaMundo`), `Name` (filled with `HolaMundo`), `Description` (filled with `Nuevo proyecto Spring Web con soporte a Thymeleaf`), and `Package name` (filled with `es.cifpcm.HolaMundo`).
- Packaging:** Includes radio buttons for `Jar` (selected) and `War`.
- Java:** Includes radio buttons for `19`, `17` (selected), `11`, and `8`.
- Dependencies:** Includes a button `ADD DEPENDENCIES... CTRL + B`. Below this, there are two sections:
 - 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."

At the bottom of the interface, there are three buttons: `GENERATE CTRL + G`, `EXPLORE CTRL + SPACE`, and `SHARE...`.

Crear Nuevo Proyecto Spring Web MVC

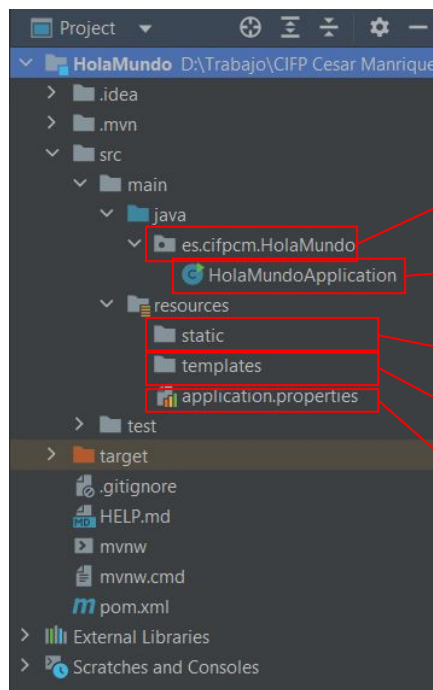
Tras ejecutar el proyecto, nos muestra en consola la URL y puerto para abrirlo en el navegador, pero al no haber definido ningún *endpoint* mostrará error.

```
342 INFO 15156 --- [main] e.cifpcm.HolaMundo.HolaMundoApplication : No active profile set, falling back to 1 default profil
952 INFO 15156 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
962 INFO 15156 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
962 INFO 15156 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.4]
981 INFO 15156 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
982 INFO 15156 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1.1s
992 WARN 15156 --- [main] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (p
527 INFO 15156 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context pat
```



Crear Nuevo Proyecto Spring Web MVC

Veamos los elementos importantes de la estructura básica de un proyecto generado:



Paquete principal del proyecto.

Contendrá subpaquetes para controladores, modelos, servicios, etc

Clase principal del proyecto identificada por la anotación **@SpringBootApplication** y que tiene el método **main**

Contendrá los elementos estáticos que usen las vistas: hojas de estilos, *scripts js*, imágenes, etc.

Contendrá las plantillas de las vistas usadas por el motor de plantillas Thymeleaf que usaremos.

Archivo donde se definirán configuraciones propias del proyecto (Ejem.: Puerto usado, uso de plantillas, etc)

Spring Web MVC

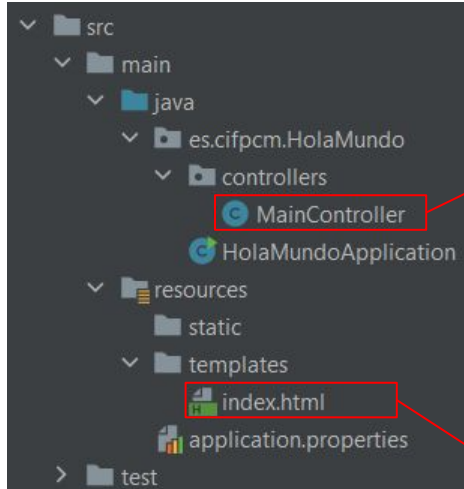
Ejemplo “Hola Mundo”

Para conseguir hacer funcionar nuestro primer ejemplo, es necesario:

- Crear una vista (archivo html) en el directorio “templates” que muestre el texto “Hola Mundo”
- Crear un nuevo controlador, en el paquete “controllers” previamente creado, que defina un *endpoint* de acceso e invoque a la vista creada en el punto anterior.

Tras estos nuevos cambios, la estructura del proyecto será la siguiente:

Spring Web MVC

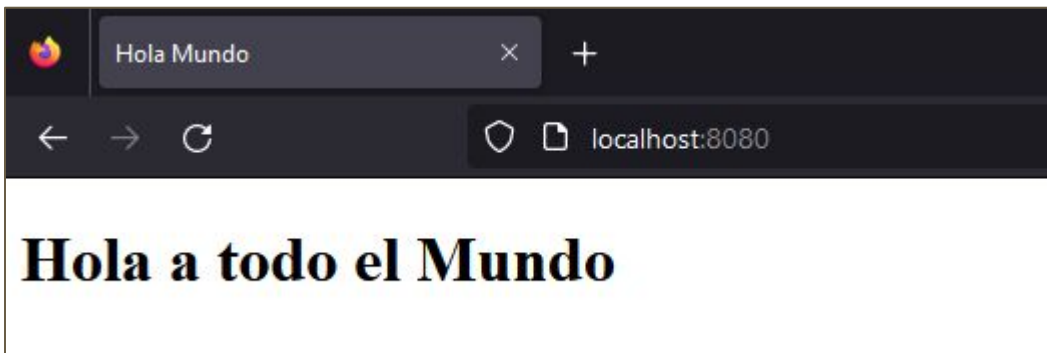


```
MainController.java
1  package es.cifpcm.HolaMundo.controllers;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.web.bind.annotation.GetMapping;
5
6  @Controller
7  public class MainController {
8      @GetMapping("/")
9      public String saluda(){
10          return "index";
11      }
12  }
```

```
index.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Hola Mundo</title>
6  </head>
7  <body>
8      <h1>Hola a todo el Mundo</h1>
9  </body>
10 </html>
```

Spring Web MVC

En este punto, tras relanzar el proyecto y acceder al *endpoint* definido en la anotación `@GetMapping("/")` veremos como se carga la plantilla y su contenido:



Spring Web MVC

Paso de datos a la vista

Para poder utilizar datos en la vista, que procedan del controlador, hay que realizar las siguientes acciones en los elementos implicados:

- Controlador:

- Recibir por parámetro un objeto de tipo "Model"
- Añadir un atributo al "Model" formado por un par clave-valor

```
@GetMapping("/")
public String saluda(Model model){
    model.addAttribute( attributeName: "saludo" attributeValue: "Hola mi gente!");
    return "index";
}
```

- Vista:

- Añadir atributo (xmlns:th="<http://www.thymeleaf.org>") en la etiqueta "html" de la plantilla
- Incluir un atributo "th:text" en el elemento que queramos mostrar el dato



```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Hola Mundo</title>
</head>
<body>
  <h1 th:text="${saludo}">Hola a todo el Mundo</h1>
</body>
</html>
```

Spring Web MVC : Controladores

Los controladores se componen principalmente de lo siguiente:

Anotación que transforma una clase simple en un controlador Spring

```
@Controller
public class MainController {
    @GetMapping("/")
    public String saluda(Model model){
        model.addAttribute("saludo", "Hola!");
        return "index";
    }
}
```

Invocación de la vista.
Se realiza un "return" del nombre de la vista sin extensión.

Anotación para métodos de un controlador que define:

- Tipo de método HTTP
- *Endpoint* de acceso

Anotaciones para otros métodos:

- GetMapping
- PostMapping
- PutMapping
- Etc.

Función para añadir atributos al modelo que será accesible desde la vista. Utiliza un sistema clave-valor y pueden incluirse objetos.

Spring Web MVC : Vistas

Las vistas se componen principalmente de lo siguiente:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Hola Mundo</title>
</head>
<body>
  <h1 th:text="${saludo}">Hola a todo el Mundo</h1>
</body>
</html>
```

Referencia al origen del motor de plantillas utilizado, en este caso Thymeleaf

Nombre del atributo del modelo que queremos acceder

Atributo propio de Thymeleaf para incluir texto en un elemento HTML. Las distintas formas de acceder a los datos desde las plantillas en Spring MVC se detallan en la [documentación de Thymeleaf](#)

Spring Web MVC : Vistas Parciales

El uso de vistas parciales nos permite reutilizar elementos que son comunes en todas las páginas (*header, footer, menu, etc*).

En *Thymeleaf* los distintos elementos reutilizables se denominan **fragmentos**.

Existen dos formas de componer las vistas con diversas partes:

- **Estilo Jerárquico**
 - Consiste en tener un *layout* principal en el que se define un área en el que se cargarán las vistas parciales correspondientes.
 - Este estilo requiere algo más de configuración.
 - Es el método que ya utilizamos en ASP.NET
- **Estilo inclusivo**
 - Se trata de definir algunos elementos como fragmentos e incluirlos en otras vistas.
 - A diferencia de ASP.NET los fragmentos tendrán que estar dentro de una página HTML completa, ya que se ha de definir el atributo “xmlns” que hace referencia a Thymeleaf en la etiqueta <html> para que pueda ser reconocida.
 - Será la manera de trabajar que utilizaremos.

Spring Web MVC : Vistas Parciales

Estilo Inclusivo

Para hacer uso de fragmentos usaremos los siguientes atributos.

- **th:fragment** : Para definir un elemento como fragmento y así poder ser utilizado en otras vistas

```
<nav th:fragment="menu">  
  <!--Contenido del Nav-->  
</nav>
```

- **th:replace** : El elemento que incluye este atributo se reemplaza por el fragmento referenciado. Se ha de especificar "directorio/archivo::nombre_fragmento"

```
<div th:replace="~{fragments/side::menu}"></div>
```

Ejercicio 1

Realiza las siguientes acciones en el nuevo controlador creado:

- Crea un nuevo método en el controlador que se despida de todo el mundo. El nombre de esta acción será “Despedida” y tendrá que cargar una vista distinta para saludar y despedirse.
- Esta acción será ejecutada para el *endpoint* “/despedida”
- Utilizaremos un fragmento para crear una cabecera común en ambas páginas (saludar y despedirse)

Spring Web MVC

Obtención de Datos de la URL

Los controladores pueden tomar datos de la URL de dos formas distintas:

- Desde un segmento de la URL
- Mediante la cadena de consulta.

Se pueden usar estas estrategias por separado o combinadas.

En los dos casos, será necesario que el método del controlador tenga definidos como parámetros de entrada los valores que desea tomar de la URL, teniendo que coincidir los nombres de los parámetros y de las variables.

Spring Web MVC

Obtención de Datos de la URL: Desde un segmento de la URL

Como ya hemos visto, la URL puede presentar datos necesarios de obtener para poder realizar las acciones necesarias en los métodos de un controlador.

En este caso, el dato puede figurar tanto al final como en el medio de la URL. Ejemplo:

- /usuarios/{userId} → para buscar un determinado usuario por su Id
- /usuarios/{userId}/facturas → para buscar las facturas de un usuario en concreto

Utilizaremos la anotación `@PathVariable` para acceder a este dato.

La URL podría tener más de un dato a recuperar.

```
@GetMapping("/usuarios/{id}")  
public String usuarios(@PathVariable String id, Model model) {
```

Spring Web MVC

Obtención de Datos de la URL: Mediante la cadena de consulta

La cadena de consulta será la parte de la URL que figura detrás del símbolo ?

Se compone por elementos clave-valor

Podrán figurar más de uno de estos pares, siempre concatenados por el símbolo &. Ejemplos:

- <http://localhost:8080/?n1=v1>
- <http://localhost:8080/?n1=v1&n2=v2>

Para recuperar estos valores utilizaremos la anotación @RequestParam de la siguiente forma:

```
localhost:8080/?name=Antonio
```

```
@GetMapping("/saludame")  
public String sayHello(@RequestParam(value = "name", required=false, defaultValue = "amigo") String name, Model model) {
```

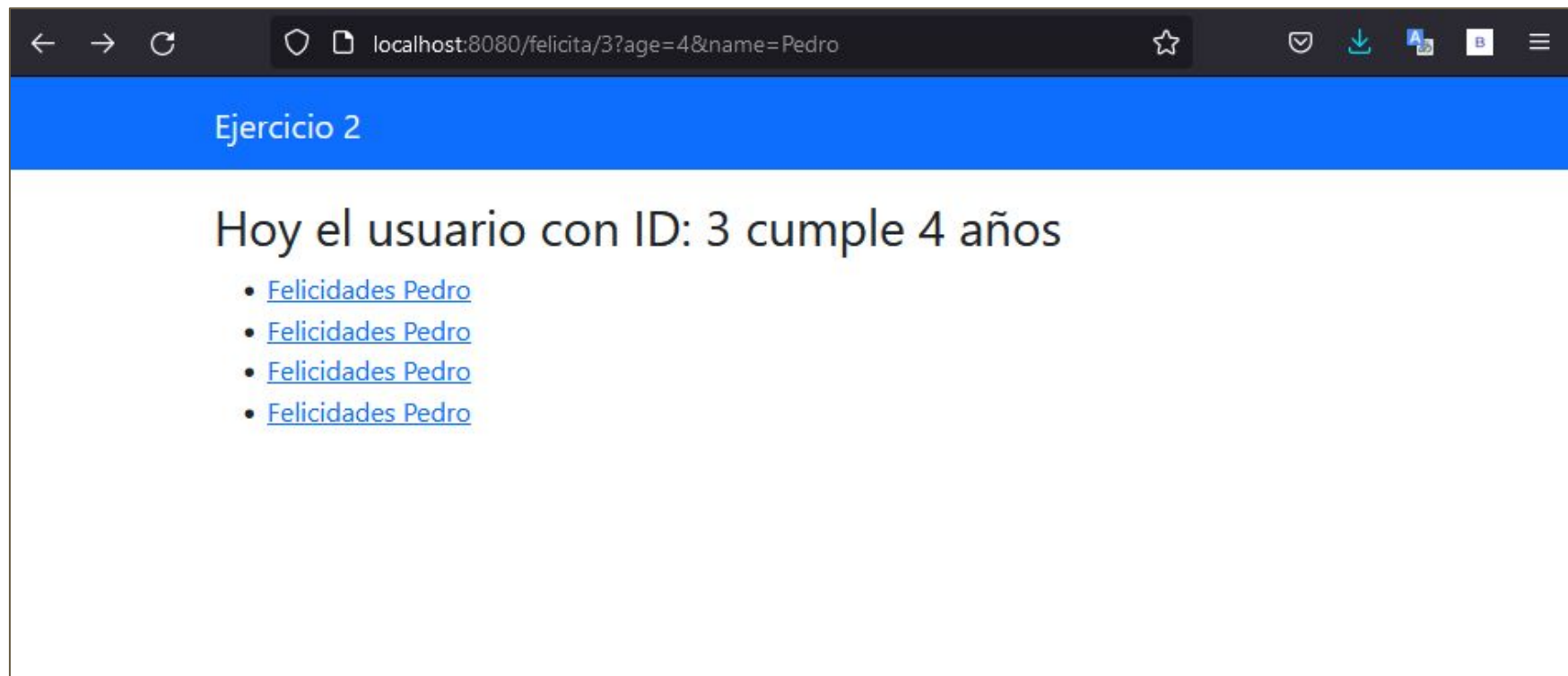
Ejercicio 2

Realizar las siguientes acciones:

- Crea un nuevo método, en tu controlador, llamada “Felicitas”
- Este nuevo método ha de recibir los siguientes datos:
 - ID de un usuario (en el tercer segmento de la URL)
 - Nombre de un usuario
 - Edad del usuario
- Crea una nueva vista, relacionada con la acción anterior, que reciba los tres datos mediante el Model y felicite al usuario tantas veces como años tenga.
- Todos esas felicitaciones tendrán que ser enlaces que naveguen a la vista de la acción por defecto del controlador “Saludos”

La apariencia de vista creada será la siguiente:

Ejercicio 2



Spring Web MVC: Modelos

En la arquitectura MVC que estamos usando, los modelos se corresponden con entidades (tablas) de nuestra base de datos.

MVC ofrece la capacidad de pasar objetos de modelos a una vista.

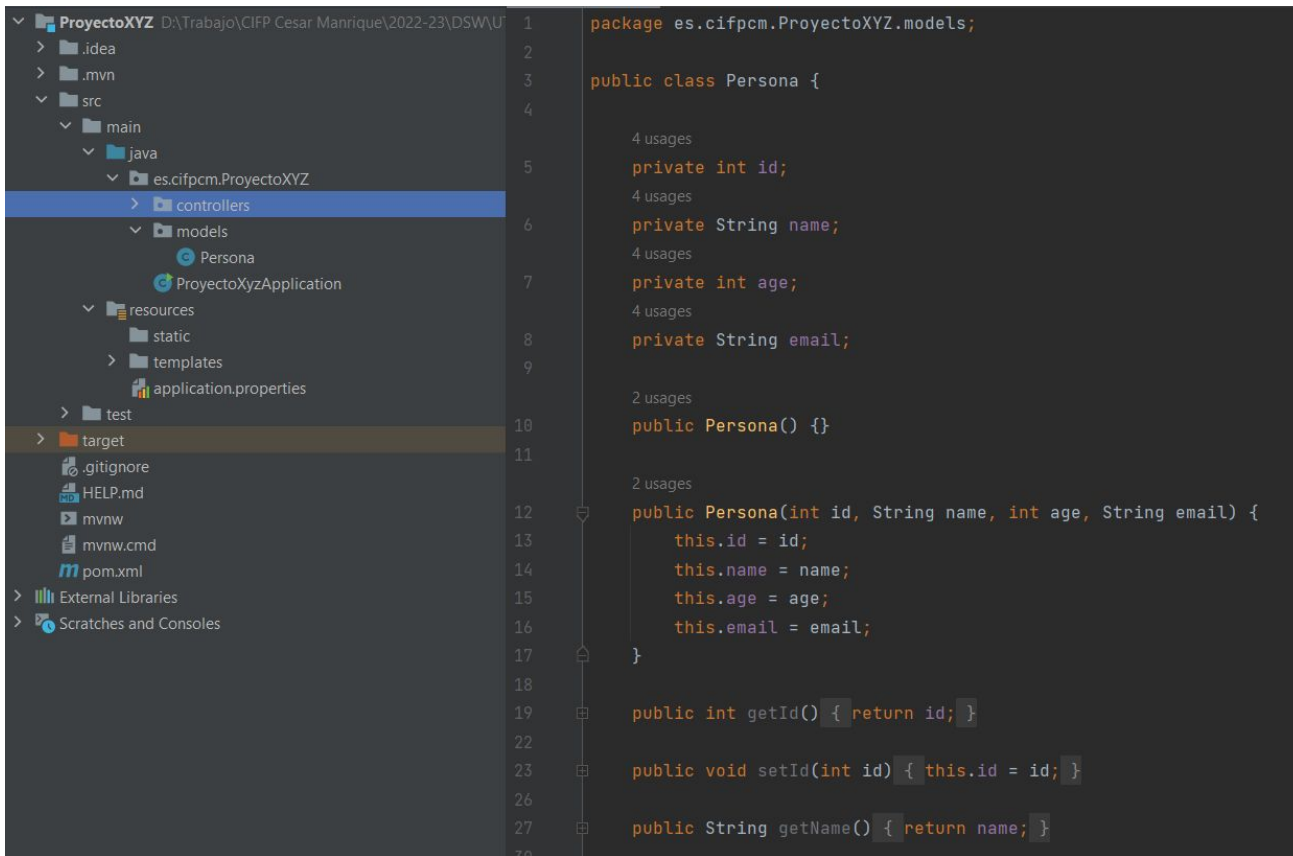
Utilizaremos los modelos como entidades para tomar datos de formularios y para persistir los datos en la BD.

Los modelos serán clases Java como las que ya conocemos (con sus atributos privados, constructores, getters/setters, etc).

A estas clases las dotaremos de anotaciones para definir comportamientos, o para facilitar la validación de estos modelos, como veremos más adelante.

Los modelos los ubicamos en un directorio "Models".

Spring Web MVC: Modelos



The image shows an IDE interface with a project explorer on the left and a code editor on the right. The project explorer displays the following structure:

- ProjectoXYZ (D:\Trabajo\CIFP Cesar Manrique\2022-23\DSW\U1)
 - .idea
 - .mvn
 - src
 - main
 - java
 - es.cifpcm.ProjectoXYZ
 - controllers (highlighted)
 - models
 - Persona (highlighted)
 - ProjectoXyzApplication
 - resources
 - static
 - templates
 - application.properties
 - test
 - target
 - .gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml
 - External Libraries
 - Scratches and Consoles

The code editor shows the following Java code for the `Persona` class:

```
1 package es.cifpcm.ProjectoXYZ.models;
2
3 public class Persona {
4
5     4 usages
6     private int id;
7     4 usages
8     private String name;
9     4 usages
10    private int age;
11    4 usages
12    private String email;
13
14    2 usages
15    public Persona() {}
16
17    2 usages
18    public Persona(int id, String name, int age, String email) {
19        this.id = id;
20        this.name = name;
21        this.age = age;
22        this.email = email;
23    }
24
25    public int getId() { return id; }
26
27    public void setId(int id) { this.id = id; }
28
29    public String getName() { return name; }
30
```

Spring Web MVC: Modelos

Enviar Modelo a la Vista

Para enviar un modelo a la vista, o un conjunto de ellos, utilizaremos la misma técnica antes vista, añadiendo un atributo con la instancia del objeto al elemento de tipo Model.

```
@GetMapping("/persona")
public String muestraPersona(Model model){

    Persona p1 = new Persona( id: 1, name: "Antonio", age: 24, email: "antonio@gmail.com");
    model.addAttribute( attributeName: "persona", p1);

    return "index";
}
```

Creación de un objeto del modelo Persona

Inclusión del objeto de tipo persona en el Model

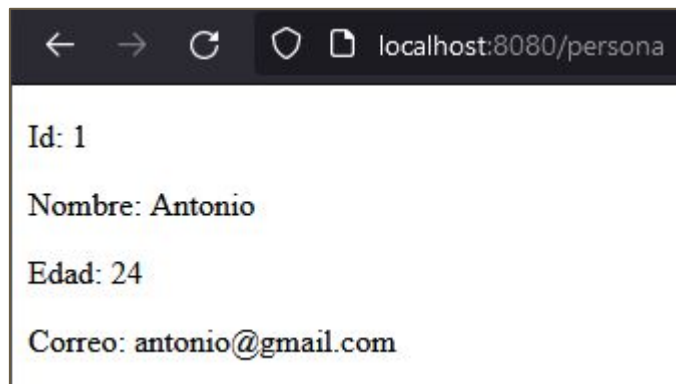
Spring Web MVC: Modelos

Mostrar Campos del Modelo en la Vista

Utilizaremos la misma técnica de ejemplos anteriores.

Para acceder a los atributos del objeto utilizaremos el operador punto (.)

```
<p th:text="${'Id: '+persona.id}"></p>
<p th:text="${'Nombre: '+persona.name}"></p>
<p th:text="${'Edad: '+persona.age}"></p>
<p th:text="${'Correo: '+persona.email}"></p>
```



Spring Web MVC: Modelos

Crear Nuevo Objeto del Modelo

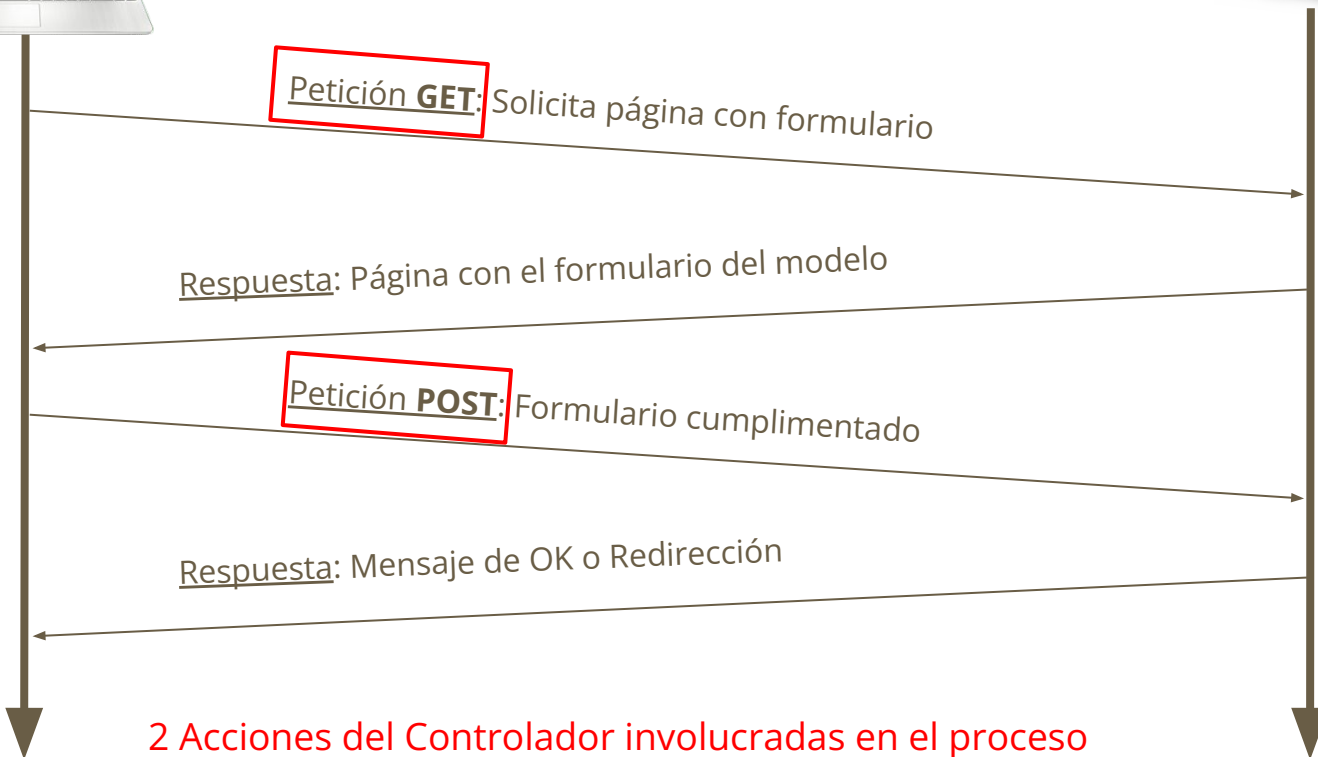
De forma general, y especialmente en las arquitecturas MPA, cuando se pretende crear un nuevo elemento de un determinado tipo de modelo, se sigue el siguiente flujo:

- El cliente comunica al servidor su intención de crear un elemento mediante una petición GET
- El servidor responde a esa petición ofreciendo una página con un formulario que contiene todos los campos necesarios para crear un elemento del modelo.
- El cliente “envía”, mediante una petición POST, todo el formulario cumplimentado
- El servidor crea el objeto del modelo solicitado, siempre y cuando todos los campos del formulario superen la validación pertinente.

Este flujo se podría representar de forma gráfica de la siguiente manera:



CREAR UN NUEVO ELEMENTO DE UN MODELO



Spring Web MVC: Modelos

Crear Nuevo Objeto del Modelo

Por tanto, son 2 acciones las involucradas en el proceso de crear un nuevo elemento de un modelo.

- 1ª Acción - GET
 - Su función es la de generar la vista con el formulario completo para crear el modelo
 - Carga una vista que utilizará el modelo para relacionar los elementos label/input con los campos del modelo
- 2ª Acción - POST
 - Recibe la información del formulario en forma de instancia del modelo con sus campos con valores
 - Se encarga de gestionar el modelo recibido → puede persistirlo en la BD o alguna otra función
 - Puede verificar que los datos recibidos sean válidos

Por convenio se utiliza el mismo nombre para estas dos acciones.

Spring Web MVC: Modelos

Crear Nuevo Objeto del Modelo - 1ª Acción

Esta acción se encarga de cargar la vista con el formulario correspondiente.

Se pasará a la vista un objeto vacío de la clase del elemento a crear. De esta forma se emparejarán los campos del formulario con los de la entidad a crear.

```
@GetMapping("/persona/create")
public String crearPersona(Model model){
    model.addAttribute( attributeName: "persona", new Persona())
    return "persona/crear";
}
```

Envío del objeto vacío a la vista mediante un nuevo atributo en el Model utilizando un nombre determinado.

Se carga al vista "crear" del directorio "persona"

Spring Web MVC: Modelos

Crear Nuevo Objeto del Modelo - 1ª Acción

El formulario de la vista ha de definir principalmente:

- La acción a la que se llamará tras enviar el formulario
- Emparejar cada campo del formulario con un campo de la entidad a crear

```
<form action="#"  
  th:action="@{/persona/create}"  
  th:object="${persona}"  
  method="post">  
  
  <label for="name"></label>  
  <input type="text" id="name" th:field="*{name}">  
  
  <input type="submit" value="Crear">  
</form>
```

Acción a invocar en el envío del formulario

Objeto a crear con el formulario y enviado en el Model

Atributo que empareja este *input* con un campo determinado de la entidad a crear

Spring Web MVC: Modelos

Crear Nuevo Objeto del Modelo - 2ª Acción

Esta acción recibe los datos del formulario en forma de modelo y realiza las acciones pertinentes en función de lo que se quiera conseguir.

```
@PostMapping("/persona/create")
```

En esta parte el método ha de ser de tipo Post

```
public String crearPersona(@ModelAttribute("persona") Persona persona){  
    //En este punto ya tenemos la persona creada  
    return "redirect:/";  
}
```

Anotación que permite recuperar el objeto creado en el formulario

Tras realizar las acciones pertinentes con el objeto obtenido (almacenar en la BD, etc) podemos redirigir la navegación de esta forma a otra acción de algún controlador

Spring Web MVC: Modelos

Validación del Modelo en su Creación o Edición

Este proceso involucra de diferente forma a varios archivos del proyecto que participan en la creación/modificación de un elemento de un modelo.

- Clase que define el modelo
- Controlador llamado al hacer *submit* del formulario
- Vista del formulario para crear el modelo

Cada uno de estos elementos tiene una labor determinante en la validación del modelo para su creación o modificación.

Para aplicar validación tendremos que incluir una nueva dependencia al POM:

Validation

I/O

Bean Validation with Hibernate validator.

Spring Web MVC: Modelos

Validación del Modelo en su Creación o Edición - Modelo

Como se comentó anteriormente, los campos de un modelo pueden ir acompañados de anotaciones de validación que definen restricciones a validar sobre cada uno de ellos. Pueden definir un mensaje de error personalizado.

Algunos ejemplos y uso de estas puede ser:

- @NotNull
- @NotEmpty
- @NotBlank
- @Min y @Max
- @Size
- @Pattern
- @Email
- y hay más...

```
@NotBlank(message = "Name is mandatory")
private String name;

@NotBlank(message = "Email is mandatory")
private String email;
```

```
@NotBlank
@Size(min = 3, max = 12)
private String username;

@NotBlank
@Size(min = 6)
private String password;
```

Spring Web MVC: Modelos

Validación del Modelo en su Creación o Edición - Controlador

La validación de datos recibidos en el servidor es una acción imprescindible, ya que es posible saltarse las validaciones del lado cliente.

En la acción (de tipo POST) que recibe el modelo con los datos del formulario se realizará la siguiente comprobación:

- Que los datos que trae el modelo generado con los datos del formulario cumple con todas las restricciones definidas en el modelo mediante los atributos de validación

Veamos un ejemplo:

Spring Web MVC: Modelos

Validación del Modelo en su Creación o Edición - Controlador

```
@PostMapping("/persona/create")
public String crearPersona(@Valid @ModelAttribute("persona") Persona persona,
                           BindingResult bindingResult){
    if(bindingResult.hasErrors()){
        //Modelo inválido
        return "persona/crear";
    }else{
        //Modelo válido
        return "redirect:<alguna_parte>";
    }
}
```

Spring Web MVC: Modelos

Validación del Modelo en su Creación o Edición - Formulario

Se adaptará el formulario para poder mostrar los posibles errores que fueran reportados por el controlador al validar el modelo recibido.

Utilizaremos atributos de Thymeleaf para modificar la apariencia del formulario para conseguir dos cosas:

- Añadir/quitar/omitir clases que definen estilos CSS en función de condiciones:
 - `th:classappend="${#fields.hasErrors('<fieldName>')} ? '<clase1>' : '<clase2>'"`
 - Se puede usar ese operador ternario para evaluar valores de algún campo del modelo y añadir/quitar/omitir otros elementos de la vista, como los atributos. Se usaría el atributo `th:attrappend="...."`
- Mostrar determinados elementos de la vista para mostrar mensajes de error en caso de que exista un error con un determinado campo, usando dos atributos:
 - `th:if="${#fields.hasErrors('<fieldName>')}" th:errors="*{<fieldName>}"`

Spring Web MVC: Otros Elementos

Servicios

En spring, un servicio es un tipo de *Bean* **definido por la anotación @Service**

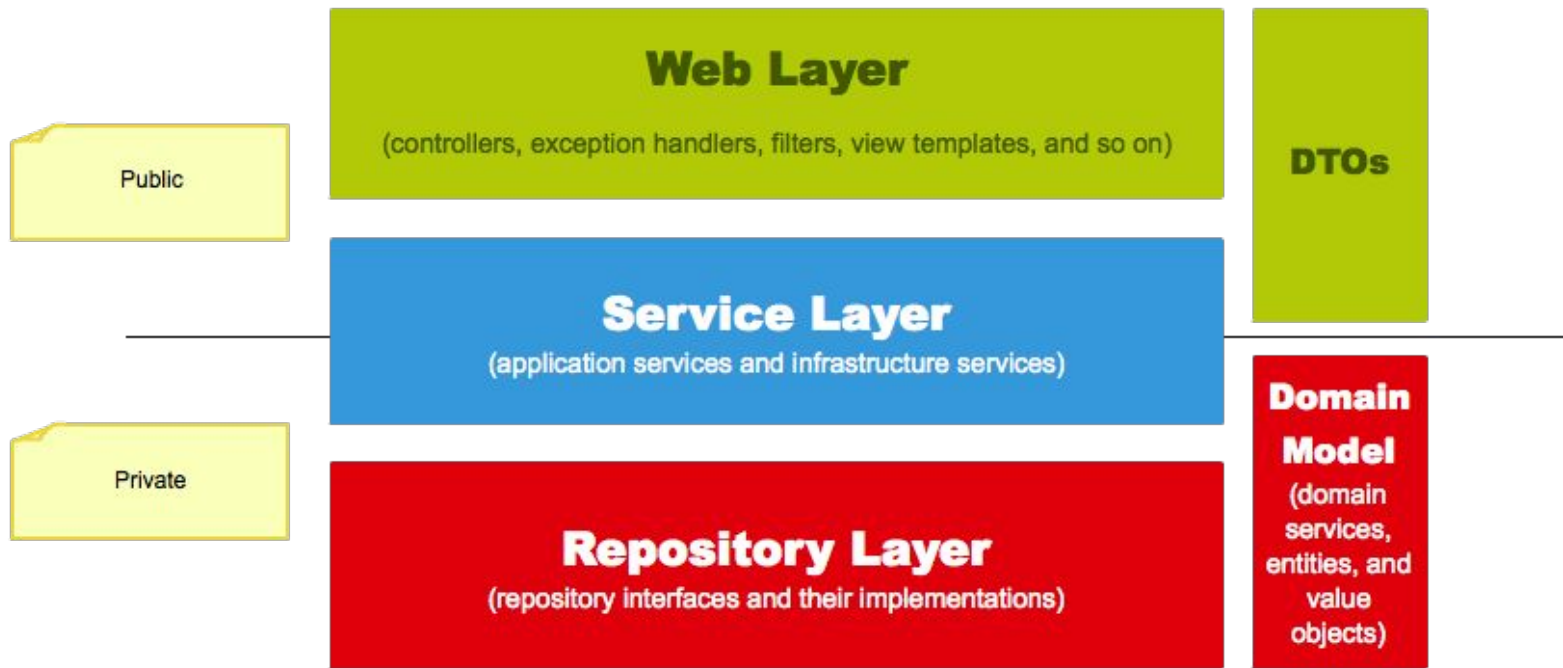
Al añadir esta anotación a una clase la dotamos de ciertas características. Una de ellas sería que **cumpliría con el patrón *singleton***.

Una de las funciones que tendrán los servicios en nuestros proyectos será el de **ofrecer una capa de abstracción a los controladores** para independizarlos de la forma de almacenamiento de los datos.

Para utilizar un servicio desde un controlador, lo declararemos (solo declarar) como atributo del controlador y lo anotaremos con *@Autowired*

Spring Web MVC: Otros Elementos

Servicios



Ejercicio 3

Generar un proyecto Spring MVC que permita realizar las siguientes acciones sobre elementos de un modelo que representa a Animales:

- Listar el conjunto de animales
- Mostrar detalles de un animal por su id
- Crear nuevos animales
- Actualizar un animal por su id
- Eliminar un animal por su Id

El Id de un animal será algo que tenga que gestionar la aplicación y por tanto no tendrá que especificarlo el usuario en ningún caso.

Ejercicio 3

El modelo que representa a estos animales tendrá que tener los siguientes atributos (con sus respectivas restricciones)

- Id
- Nombre (campo obligatorio, min. 3 caracteres y máx. 15)
- VidaMedia (campo obligatorio, numérico, valor mínimo 0 y máximo valor 600)
- Extinto (tipo de dato lógico y obligatorio)

La creación de nuevos animales requiere de validación de los campos en el cliente, adaptando el formulario a los requisitos, y en el servidor.

Todos los textos de la página tendrán que aparecer en castellano (nombres y validaciones).

Se tendrá que usar *Bean* de Servicio para gestionar la simulación de persistencia de los datos, como hemos visto en clase.