DAW
Desarrollo de Aplicaciones Web
2º Curso

DWES
Desarrollo Web Entorno Servidor

UD 5 Spring Boot

2. Spring MVC

IES BALMIS
Dpto Informática
Curso 2025-2026
Versión 2 (12/2025)

UD5.2 – Spring MVC

ÍNDICE

1. Conceptos clave JAVA EE

Java EE (Java Platform, Enterprise Edition) es una plataforma de desarrollo para crear aplicaciones empresariales en Java. Se basa en el lenguaje de programación Java y ofrece una serie de especificaciones, API y herramientas para facilitar el desarrollo de aplicaciones escalables, seguras y que pueden funcionar en entornos distribuidos.

Java EE incluye varias APIs y especificaciones importantes, como:

- **Servlets**: Para manejar solicitudes y respuestas en aplicaciones web.
- **JavaServer Pages (JSP)**: Para crear contenido dinámico en la web.
- **Enterprise JavaBeans (EJB)**: Para desarrollar componentes de negocio distribuidos en aplicaciones empresariales.
- **Java Persistence API (JPA)**: Para gestionar la persistencia y la interacción con bases de datos.

Java EE proporciona **contenedores**, es decir, entornos ejecutivos donde las aplicaciones se ejecutan. Estos contenedores gestionan todos los aspectos del ciclo de vida de las aplicaciones, incluida la seguridad, la transacción y la gestión de recursos.

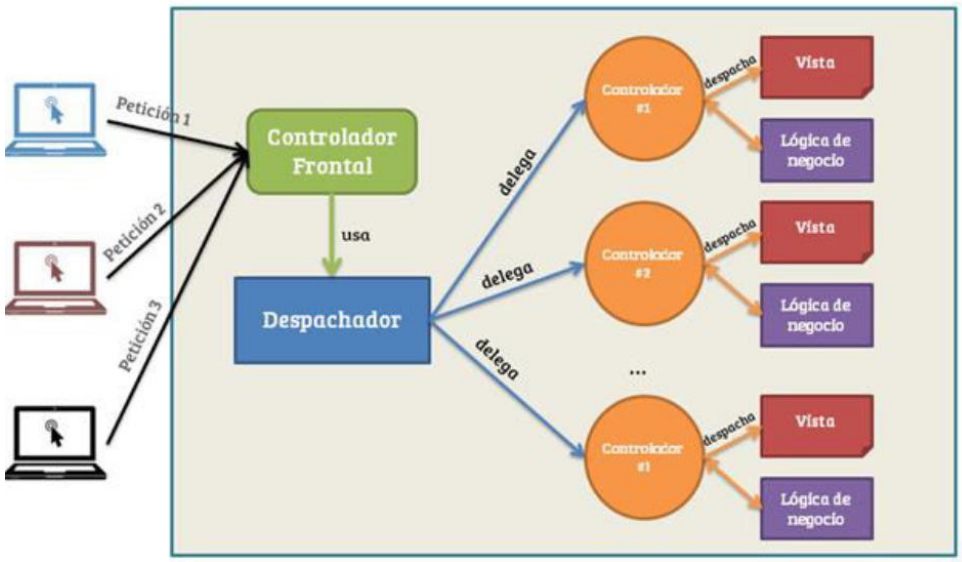
Los Servlets que proporciona Java EE se definen por:

- Son una clase que nos permite **gestionar peticiones/respuesta** de una aplicación servidor (normalmente mediante un Controlador)
- Usualmente usados en contextos web bajo el **protocolo HTTP**
- Se ejecutan en un **contenedor de web** (no es lo mismo que servidor web)

Un **contenedor web** maneja el ciclo de vida de dichos componentes, gestiona sesiones de usuario, proporciona administración de recursos (como conexiones a BD) y **procesa las solicitudes HTTP.** Después de analizar las solicitudes HTTP, devuelve los archivos solicitados o las dirige a los servlets vinculados a la URL, ejecutando su código y devolviendo las respuestas a los clientes. Esta tarea se desarrolla gracias al **Dispatcher Servlet**.

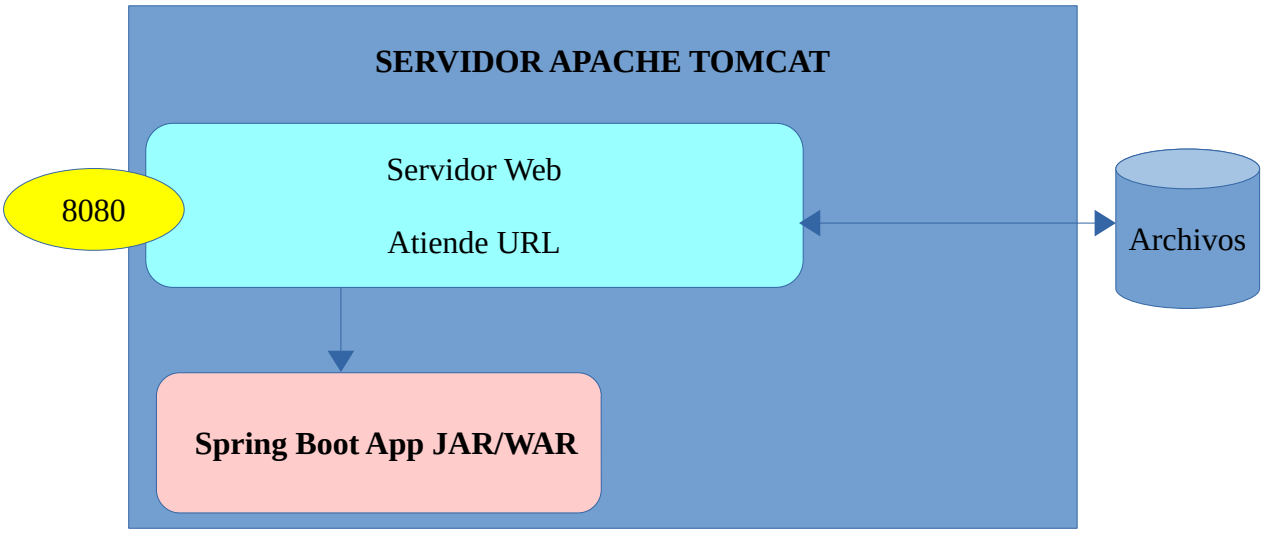
El **Front Controller** es un patrón de diseño arquitectónico utilizado en aplicaciones web para manejar todas las solicitudes entrantes a través de un único controlador centralizado. Este enfoque permite una organización más clara y un mayor control sobre el flujo de la aplicación, facilitando la implementación de diversas funcionalidades como la gestión de seguridad, el enrutamiento y la gestión de sesiones.

De esta forma se centraliza la gestión de peticiones, se aumenta la reusabilidad y se mejora la seguridad, aunque puede disminuir la velocidad por la propia centralización.



Spring Boot implementa el Despachador (Dispatcher Servlet) de forma transparente este Front Controller redirigiendo la petición a cada controlador analizando la URL.

Las aplicaciones Spring se instalan en servidor web con Java EE y su URL base es única. El esquema general de instalación de una aplicación Spring en un servidor Apache Tomcat que incluye Java EE sería:



Spring puede generar aplicaciones **JAR** o **WAR**.

Generaremos un proyecto Spring de tipo **WAR** cuando deseamos desplegar/instalar una aplicación en un Servidor Web con Java EE, mientras que usaremos **JAR** cuando, generalmente en desarrollo, deseamos crear en tiempo de ejecución servicios añadidos para probarla como:

- Crear un servidor Apache Tomcat donde ejecutar temporalmente nuestra aplicación
- Crear un servidor de Bases de Datos de tipo SQL como H2 Database para poder acceder a datos de forma persistente.

Dependencias

Al crear un proyecto de **Spring**, se cargarán una serie de dependencias (archivos JAR con componentes de software) utilizando el repositorio de **Maven**.

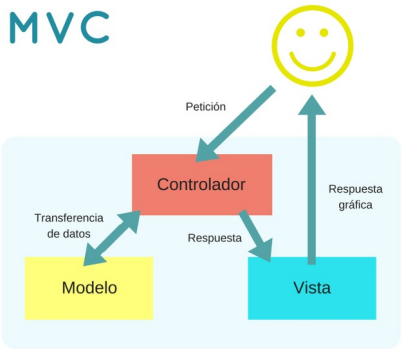
Dependiendo de la versión de **Spring Boot** (indicada en la etiqueta **<parent>** del archivo **pom.xml** de **Maven**), las dependencias utilizadas de los componentes son diferentes y necesitarán una versión de Java mínima. Por ejemplo:

Spring Boot	Spring FrameWork (Core)	Java
4.0.x	**7.0.x**	**21**
3.5.x	**6.2.x**	**21**
3.4.x	**6.2.x**	**21**
3.3,x	**6.1.x**	**17**

Podemos ver los archivos JAR descargados por nuestros proyectos en la cache de Netbeans en:

```
.\NetBeans-Portable-27\portable-config
                    \maven_repository\org\springframework\spring-core
```

Anotaciones



Las anotaciones en Spring (y Spring Boot) son la base de la configuración declarativa, la inyección de dependencias y la configuración automática que hacen que el framework sea tan potente, flexible y fácil de usar.

Spring sigue el patrón Modelo-Vista-Controlador que describiremos en detalle posteriormente y sigue el esquema aquí mostrado.

Las piezas fundamentales de Spring son sus componentes ya que al iniciar la aplicación, la clase principal identificada con **@SpringBootApplication** realiza un scan buscando todas las clases identificadas como componentes **@Component**.

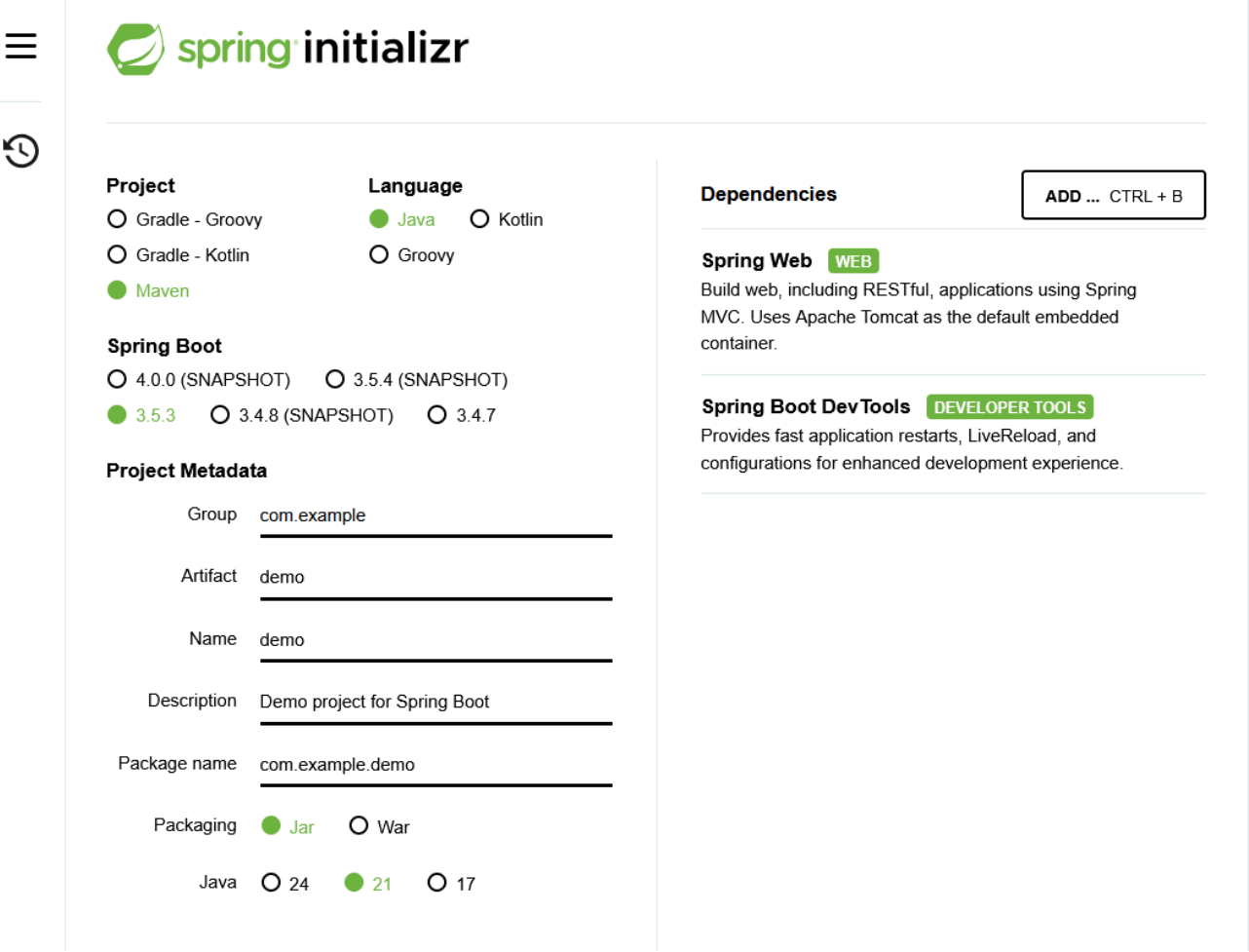
Los componentes más importantes (ya que que heredan de **@Component)** son:

@Controller	Clase controlador que atiende, procesa y redirige la salida. Es el Controlador en el modelo MVC.
@Service	Clase que define el interfaz del modelo de negocio. Es el Modelo en el modelo MVC. Esta clase utilizará otras clases imprescindibles dentro del Modelo como **@Repository** y **@Entity** que veremos más adelante.

2. Generar proyecto base de Spring MVC

Para generar un proyecto Spring podemos usar el asistente web ***Spring Initializr***

http://start.spring.io



Dejaremos la versión estable propuesta por Spring Boot, ya que las superiores son SNAPSHOT.

En desarrollo usaremos JAR para disponer de un Apache Tomcat embebido en tiempo de ejecución. Más adelante cambiaremos de JAR a WAR para desplegar estos proyectos en un Apache Tomcat de producción.

Seleccionaremos **Java 21** porque es la versión LTS actual, y es la que tenemos instalada en el instituto.

Añadiremos las dependencias:

- Spring Web
- Spring Boot Dev Tools

Thymeleaf (no es necesario todavía)

NetBeans: Abrir y compilar el proyecto

Desde NetBeans con el plugin NB SpringBoot instalado, debemos abrirlo y compilarlo para que descargue las dependencias y cree la estructura del proyecto.

En este proceso se creará un archivo **nbactions.xml**. Si el proyecto lo hemos creado con el asistente del propio NetBeans ya tendremos este archivo, pero habrá creado acciones predeterminadas que podemos consultar en **Properties → Actions**. Si no deseamos esta configuración predeterminada, podemos eliminarlo y volver a compilar para que se genere, lo que nos dejará acceder a la configuración por defecto de **Properties → Run** como en los proyectos de consola.

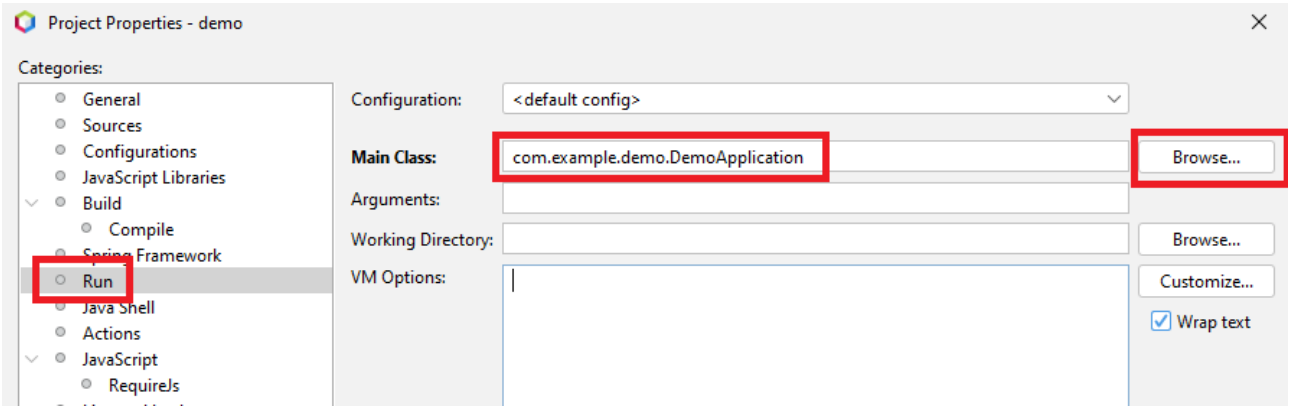
Si el proyecto se ha creado como copia de un proyecto anterior, para eliminar antiguos archivos de Maven almacenados por NetBeans debemos compilar con Maven Clean Install desde:

Properties → Run Maven → Clean Install

NetBeans: Ejecutar el proyecto

Para hacer un proyecto **HolaMundo** podemos comenzar por abrir el proyecto **demo** creado anteriormente.

Si hemos creado el proyecto desde el asistente web ***Spring Initializr*** debemos ir a **Properties → Run e** indicar el Main Class:



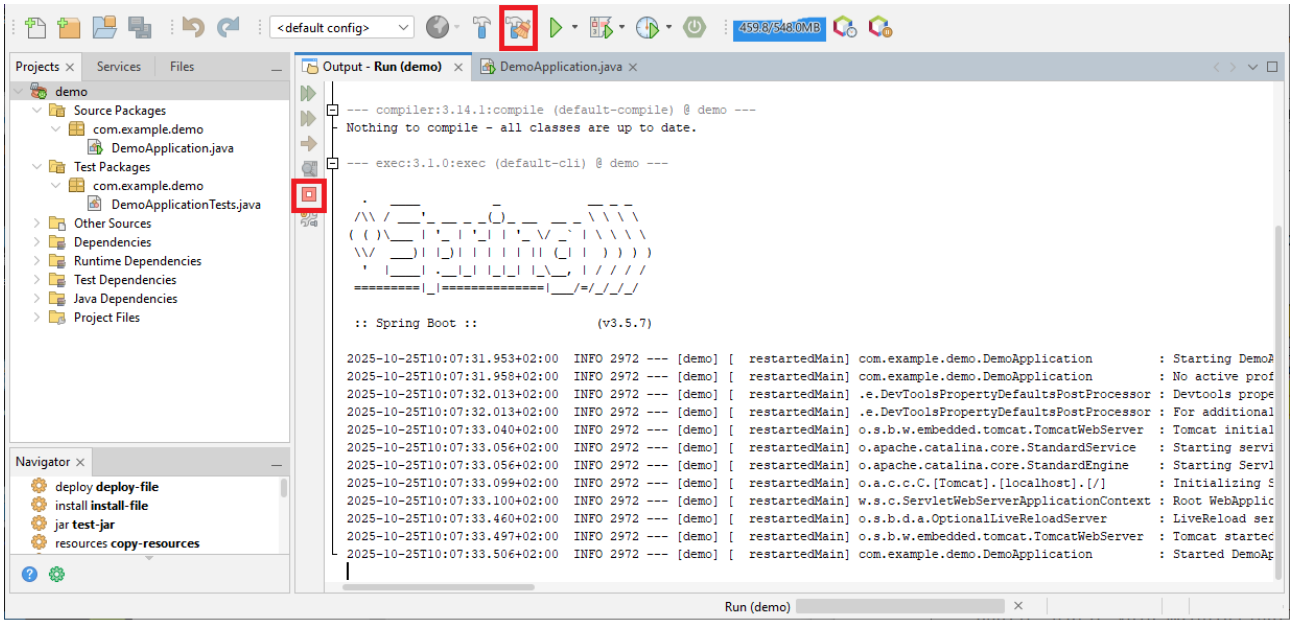
El plugin NB Spring Boot de NetBeans ya realiza este paso.

El proyecto demo todavía no realiza ninguna acción, por lo que para mostrar en el navegador "**Hola Mundo!**" debemos editar la clase que contiene el main.

DemoApplication.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@SpringBootApplication
@Controller
public class DemoApplication {

    @ReponseBody
    @GetMapping("/hola")
    public String hola() {
        return "Hola Mundo!";
    }

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

Al compilar y ejecutar se pondrá en marcha un Servidor Web Apache Tomcat con la aplicación ejecutándose y quedando a la espera de peticiones desde el navegador.



Para probarla, abriremos el navegador con la URL:

http://localhost:8080/hola

IMPORTANTE:

- Al compilar con martillo-escoba se actualiza la aplicación en el servidor de **Apache Tomcat embebido** si está ya en ejecución
- Para detener la aplicación (y el Servidor Web) pulsar en el botón rojo de la ventana **Output**
- Las anotaciones **@ReponseBody** y **@GetMapping** las veremos más adelante

3. Contenido web estático

3.1 Proyecto demo-static

Desde NetBeans vamos a crear un proyecto **demostatic** con el asistente.

La carpeta de archivos HTML+CSS+JS para el servidor web (lo equivalente a un Servidor Web como Apache) del proyecto para el servidor web es **src/main/resources/**, aunque en **NetBeans** se muestra como **Web Pages.**

La dependencia **Spring Web**, crea un Apache Tomcat embebido en el proyecto que se ofrece por defecto en el puerto **8080**.

Configuración del Context Path

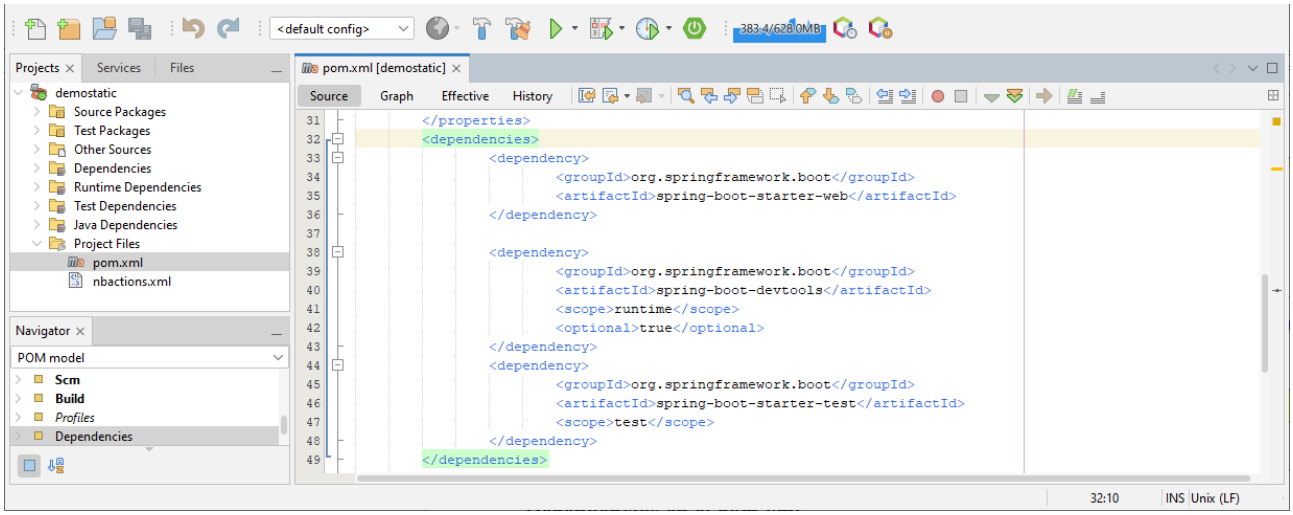
En Spring Boot debemos cambiar el valor por defecto de variables de configuración en el archivo **src/main/resources/application.properties** para establecer el **context path** de este proyecto de SB (Spring Boot):

src/main/resources/application.properties

```
spring.application.name=demostatic
server.servlet.context-path=/demostatic
```

Dependencias y configuración en el pom.xml

El archivo **pom.xml** contendrá las dependencias indicadas al crearlo.
Este archivo se puede editar, y por lo tanto podemos añadirle posteriormente más dependencias para disponer de más funcionalidad.



También podremos **consultar y/o editar** en el mismo fichero **pom.xml** la versión de **Java** indicada y la de **Spring**. Generalmente, se suele actualizar indicando versiones más modernas que hayan aparecido, aunque si hay diferencias importantes es posible que nos obligue a cambiar algo de código.

Contenido a mostrar HTML+CSS+JS

Si creamos/copiamos un archivo **index.html** en **src/main/resources/static** se mostrará directamente en http://localhost:8080/demostatic ya que es la URL indicada en el valor de **context path** del proyecto.

Si nuestra web contiene varios archivos **.html** o archivos **.css/.js**, deberemos copiarlos también. Como ejemplo crearemos un archivo **index.html** para nuestro proyecto **demostatic.**

src/main/resources/static/index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>DEMO</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <h1>DEMO STATIC</h1>
        <div>
            <p>Este archivo html es estático</p>
        </div>
    </body>
</html>
```

Configurar la clase principal y la codificació (Spring Initializr)

En un proyecto creado con ***Spring Initializr,*** debemos abrir las propiedades del proyecto pulsando con el botón derecho y seleccionar Run. En esta pantalla:

- pulsaremos en **Browse** para seleccionar la clase que dispone del método **main**
- **(OPCIONAL)** añadiremos en **VM Options** las variables para javac que forzarán al proyecto a usar UTF-8 como codificación y salida.
  - **-Dsun.stdout.encoding=UTF-8**

Configurar la clase principal y la codificació (asistente de NetBeans)

(OPCIONAL) En un proyecto creado con el **plugin/asistente de NetBeans** podemos eliminar el archivo **nbactions.xml** y estaremos como con **Spring Initializr**, o abrir las propiedades del proyecto pulsando con el botón derecho y seleccionar

- Properties → Actions → Run project → Set Properties
- Properties → Actions → Debug project → Set Properties

Luego añadiremos en el textarea:

```
spring-boot.run.jvmArguments=-Dsun.stdout.encoding=UTF-8 -
Dsun.stderr.encoding=UTF-8 -Dfile.encoding=UTF-8
```

Esto añadirá en el archivo **nbactions.xml**:

```
<spring-boot.run.jvmArguments>-Dsun.stdout.encoding=UTF-8 -
Dsun.stderr.encoding=UTF-8 -Dfile.encoding=UTF-8</spring-
boot.run.jvmArguments>
```

Ejecutar y probar desde el navegador

Para probar, se debe ejecutar el proyecto y abrir el navegador en la web:

http://localhost:8080/demostatic

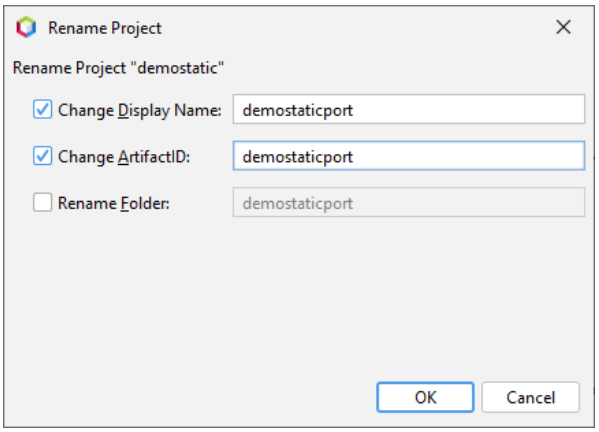
Error de página no encontrada

Cuando una página no se encuentra, da un error 404.
Para gestionarlo, podemos crear una página 404 en:
/static/error/404.html
y **Spring** la mostrará si no se encuentra el archivo indicado en la URL.

Puedes comprobar el en proyecto demostatic404.

3.2 Proyecto demo-static-port

Realizaremos una copia de nuestro proyecto **demostatic** como **demostaticport**



Una vez copiado, como el nombre del proyecto sigue siendo el mismo, debemos renombrar como se indica en esta ventana lateral.

Para cambiar las propiedades de Spring en un proyecto tenemos dos opciones:

1. En el archivo **application.properties**
2. En el método **main** en tiempo de ejecución

Deseamos cambiar el puerto en el que se inicia el servidor web Apache Tomcat del archivo JAR de Spring en nuestra aplicación.

Variables SPRING en application.properties

src/main/resources/application.properties

```
spring.application.name=demostaticport
server.servlet.context-path=/demostaticport
server.port=9000
```

Al compilar y ejecutar veremos que el puerto de la URL de nuestro proyecto ha cambiado:

http://localhost:9000/demostaticport

Variables SPRING en tiempo de ejecución

Crearemos un **Map<String,String>** con los valores a cambiar y se lo añadiremos al ejecutarlo. El nuevo método **main** quedará:

src/main/resources/application.properties

```
spring.application.name=demostaticport2
```

DemoApplication.java

```
import java.util.HashMap;
import java.util.Map;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemostaticApplication {

    public static void main(String[] args) {
        Map<String, Object> props = new HashMap<>();
        props.put("server.port", "8081");
        props.put("server.servlet.context-path", "/demostaticport2");

        SpringApplication app = new SpringApplication(DemostaticApplication.class);
        app.setDefaultProperties(props);
        app.run(args);

    }

}
```

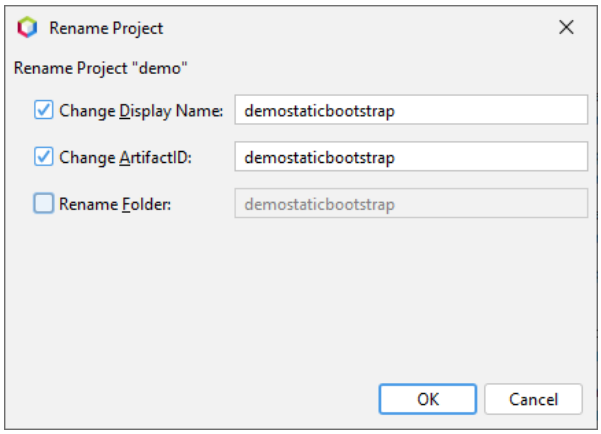
En este ejemplo cambiamos en tiempo real el puerto y el context path quedando:

http://localhost:8081/demostaticport2

Puedes comprobar el en proyecto demo-static-port2.

3.3 Proyecto demo-static-bootstrap

Realizaremos una copia de nuestro proyecto **demostatic** como **demostaticbootstrap** Una vez copiado, como el nombre del proyecto sigue siendo el mismo, debemos renombrar:

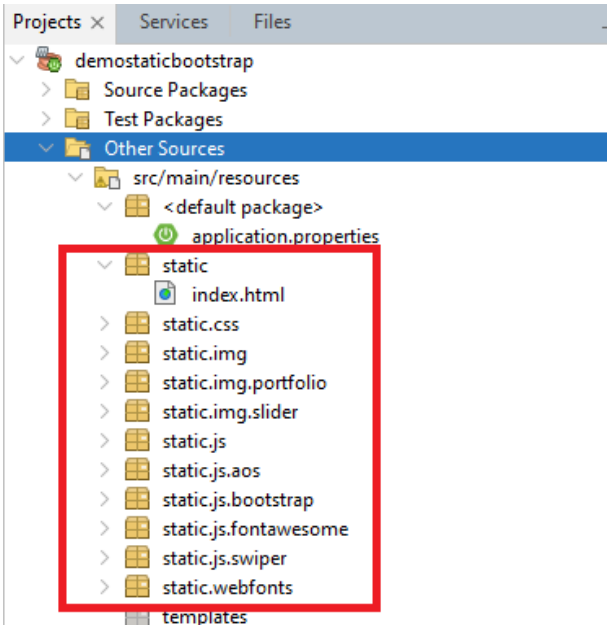


Después de copiar y renombrar, como siempre, cambiaremos las variables del proyecto de Spring Boot en **applications.properties**:

src/main/resources/application.properties

```
spring.application.name=demostaticbootstrap
server.servlet.context-path=/demostaticbootstrap
```

A continuación copiaremos un proyecto de bootstrap con todos sus archivos .html, .css y .js a la carpeta **/src/main/resources/static**.



En este ejemplo se ha copiado el contenido estático visto en el ejercicio **41html-bootstrap-local** en el apartado de PHP.

Todos los archivos son locales por lo que no se necesita internet para descargar ningún recurso solicitado por **index.html**.

Se puede probar con la URL:

http://localhost:8080/demostaticbootstrap/

3.4 Proyecto demo-static-webjars

Como vemos, la dependencia de recursos de terceros crea un **problema** de **mantenimiento** y de **disponibilidad** tanto si los usamos de forma local como con enlaces CDN:

- **mantenimiento con archivos en local**: al cambiar las versiones es necesario descargarlas y actualizar las carpetas de static
- **mantenimiento con CDN**: al cambiar las versiones es necesario cambiar el código de html, ya que en la URL del link aparecen.
- **disponibilidad con CDN**: si usamos estos recursos dependemos de un sitio web externo, por lo que si falla no obtendremos ese archivo.

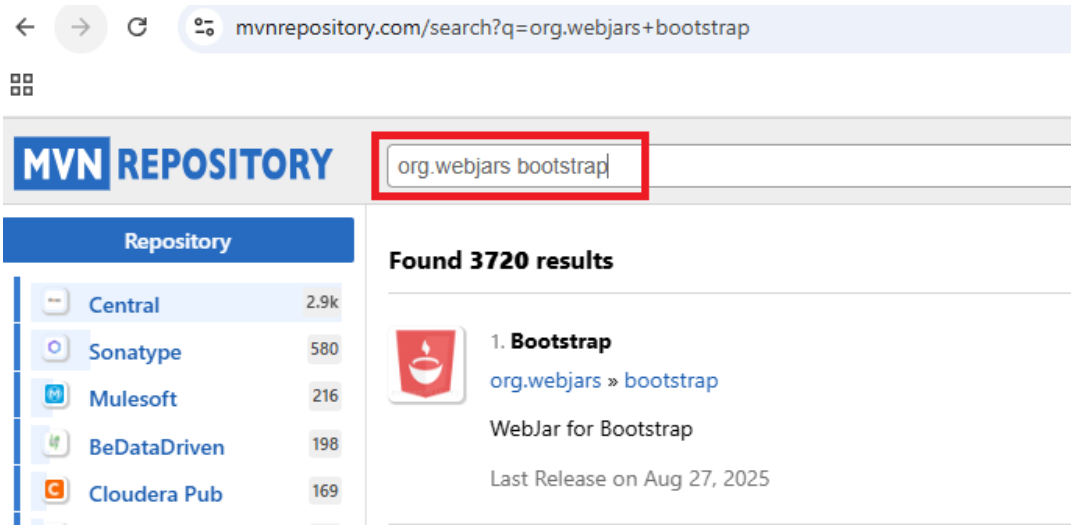
Realizaremos una copia de nuestro proyecto **demostaticbootstrap** como **demostaticwebjars.** Una vez copiado, como el nombre del proyecto sigue siendo el mismo, debemos renombrar también el **Display Name** y el **Artifact** como en ejemplos anteriores.

Para solucionar estos problemas existen archivos **.jar** con las librerías y extensiones que pueden incluirse en el proyecto a través del **pom.xml**, de forma que estén disponibles en local para no depender de un dominio externo (CDN).

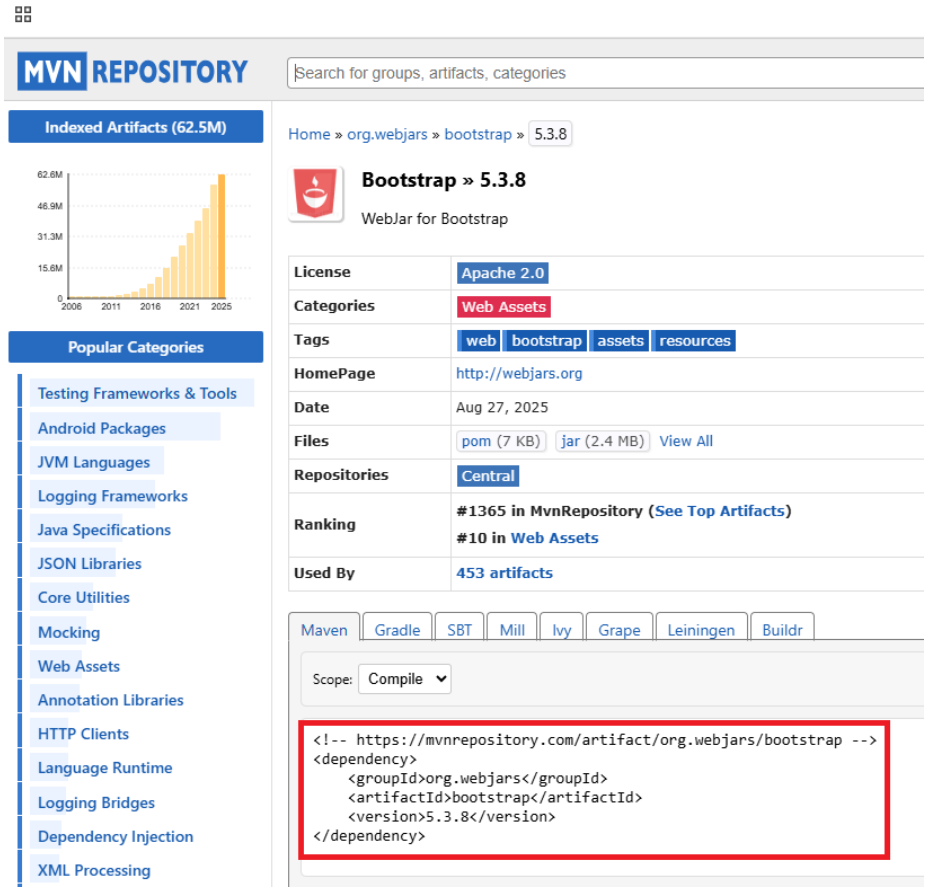
Webjars es la más utilizada y dispone de las librerías más usadas.

https://www.webjars.org/

Para incluir estas librerías, podemos buscar en **mvnrepository.com** la librería precediendo de **org.webjars**. Por ejemplo para buscar **bootstrap**:



Una vez encontrada, incluiremos el código de **maven** en nuestro **pom.xml**:



En el ejemplo hemos incluido **bootstrap, jquery, swiper y font-awesome**.
Hemos comprobado que **aos** no está incluida en **org.webjars** y la hemos encontrado en **org.mvnpm**.

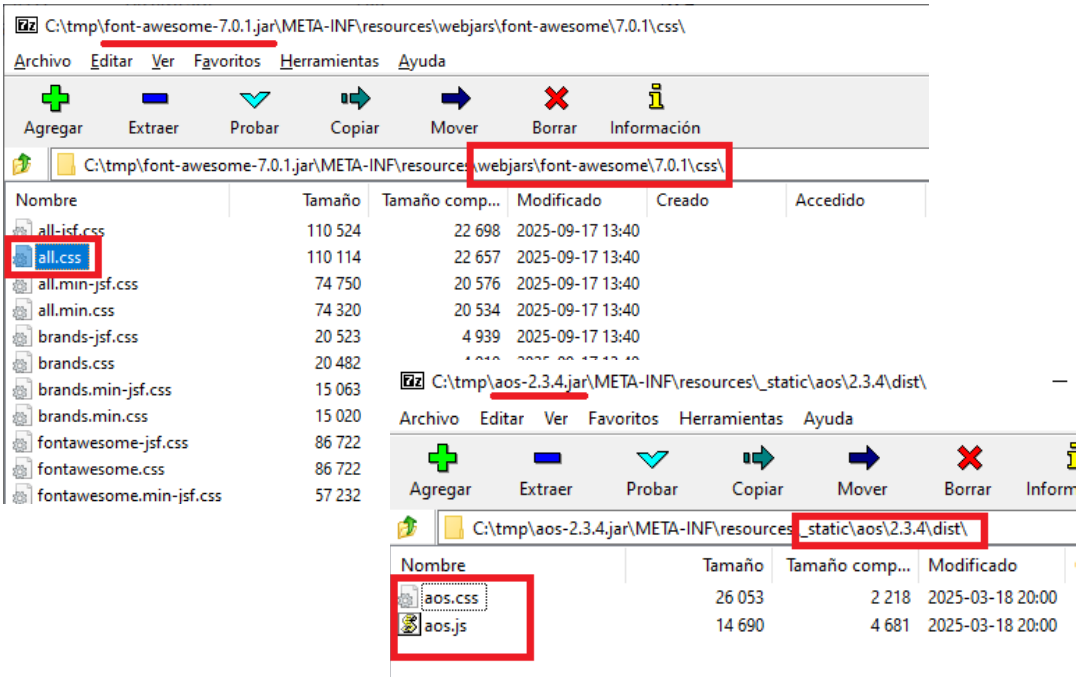
Esto soluciona parte del problema, la disponibilidad, pero no el mantenimiento, ya que seguimos teniendo la versión en el **link**.
Para evitar esto, incluiremos otra dependencia en el **pom.xml**:

pom.xml

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator-core</artifactId>
    <version>0.59</version>
</dependency>
```

Esta extensión permite eliminar la versión del **link**, ya que busca la más actual que haya disponible.

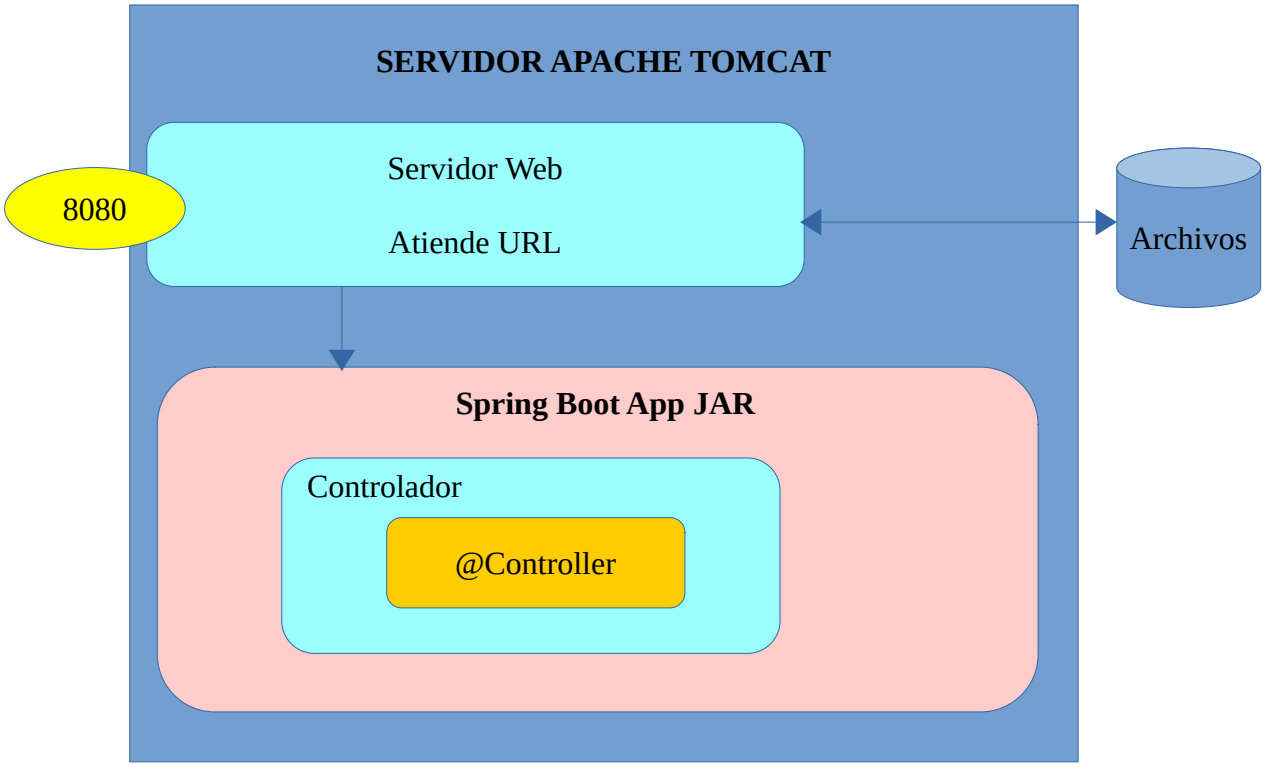
Para conocer la ruta completa de nuestro recurso incluido en el JAR, podemos abrirlo con una utilidad de archivos ZIP (7zip o winrar) y ver la ruta.



Si son **webjars** y hemos incluido **webjars-locator-core** podremos eliminar la versión en el link.

4. Controlador

Cuando añadimos un controlador a nuestro proyecto Spring podemos devolver datos generados dinámicamente o redirigir la petición HTTP a diferentes partes de la aplicación. Es esquema es:



Cuando una aplicación Spring tiene controladores, la aplicación realiza el siguiente proceso en este orden para devolver la salida:

1. Existe un controlador que atiende la URL:
   a) Puede devolver **contenido generado dinámicamente**
   b) Redirige la salida a un módulo que genere la VISTA (P.e. **Thymeleaf**)
   c) Redirige la salida a un archivo existente en **static**
2. Busca archivos en **static** que respondan a la URL
3. Muestra **error**

4.1 Proyecto demo-dinamic

Vamos a crear un proyecto **demodinamic** con varios controladores que realicen contenido generado dinámicamente. Crearemos un Java Package **controller** para los controladores.

Para que el controlador atienda mediante un método a una URL, debe indicarse con una anotación antes de dicho método.

ControladorFichero: devolviendo fichero

```
@Controller
public class ControladorFichero {

    @RequestMapping(path = "/rutafichero", method = RequestMethod.GET)
    public String metodo() {
        return "fichero.html";
    }

}
```

Al recibir el controlador de Spring
la URL http://localhost:8080/demodinamic/**rutafichero**
ejecutará el método **metodo()** devolviendo "**fichero.html**"
por lo que buscará en la carpeta **static** un archivo denominado **fichero.html**

Si queremos devolver contenido en vez de un fichero, debemos indicarlo mediante **@ResponseBody**

Controlador 1 devolviendo contenido

```
@Controller
public class ControladorString1 {

    @ResponseBody
    @RequestMapping(path = "/rutastring1", method = RequestMethod.GET)
    public String metodo() {
        return "Hola Mundo 1!";
    }

}
```

Si todos los métodos de una clase devuelven contenido y no ficheros podemos indicarlo en la clase:

Controlador 2 devolviendo contenido

```
@Controller
@ResponseBody
public class ControladorString2 {

    @RequestMapping(path = "/rutastring2", method = RequestMethod.GET)
    public String metodo() {
        return "Hola Mundo 2!";
    }

}
```

O utilizar una combinación: **@RestController** = @Controller + @ResponseBody

Controlador 3 devolviendo contenido

```
@RestController
public class ControladorString3 {

    @RequestMapping(path = "/rutastring3", method = RequestMethod.GET)
    public String metodo() {
        return "Hola Mundo 3!";
    }

}
```

También podemos abreviar la anotación **@RequestMapping** con anotaciones que incorporan el método:

@GetMapping	@RequestMapping con método GET
@PostMapping	@RequestMapping con método POST
@PutMapping	@RequestMapping con método PUT
@DeleteMapping	@RequestMapping con método DELETE
@PatchMapping	@RequestMapping con método PATCH

Nuestro ejemplo quedaría:

Controlador 4 devolviendo contenido

```
@RestController
public class ControladorString4 {
    @GetMapping("/rutastring4")
    public String metodo() {
        return "Hola Mundo 4!";
    }
}
```

Ahora crearemos una clase **DinamicController** dentro también del Java Package **controller.**

Este controlador tiene un método que crea en tiempo real un contenido String y lo devuelve como **@ResponseBody**.

DinamicController

```
@Controller
public class DinamicController {
    @ResponseBody
    @GetMapping("/dinamico")
    public String dinamico() {
        String salida = html;
        salida=salida.replace("[CONTENIDO]", "<p>Este contenido html es dinámico</p>" );
        return salida;
    }
}
```

Las URL tienen un formato que se divide en varias partes.
Veamos el formato general:

http://dominio:puerto/target/path?queryString

Dominio	Es el nombre del dominio donde se aloja el sitio web. En nuestro caso es **localhost**, que equivale a la IP 127.0.0.1
Puerto	Es el puerto de escucha del Servidor Web JavaEE. Si no lo indicamos en la URL, el valor por defecto es el 80.
Target	Es el path asignado a nuestra aplicación. En Spring es el valor del **context path** establecido en **application.properties**
Path	Es la cadena indicada en @RequestMapping o sus anotaciones heredadas. Por ejemplo: para un path=ventas en un método GET sería: **@GetMapping("/ventas")**
QueryString	Es una cadena (String) que contiene pares de variable=valor separados por el carácter &. Ejemplo: http://localhost:8080/miapp/ventas?**enero**=**1230**&**febrero**=**860**

En Spring, para poder recuperar un valor a través de un parámetro en la URL indicado en QueryString usaremos la anotación **@RequestParam** y crearemos una variable como parámetro del método con el mismo nombre de la variable.

Para nuestro ejemplo con la URL:

http://localhost:8080/demodinamic/paramquery?param=saludo

DinamicController

```
@Controller
public class DinamicController {
    @ResponseBody
    @GetMapping("/paramquery")
    public String getMethodQuery(@RequestParam String param) {
        String salida = html;
        salida=salida.replace("[CONTENIDO]", param);
        return salida;
    }
}
```

El parámetro por defecto es obligatorio, por lo que nos dará error con la URL:

http://localhost:8080/demodinamic/paramquery

En la anotación **@RequestParam** podemos indicar si es obligatorio (con required) y/o un valor por defecto.

DinamicController

```
@Controller
public class DinamicController {
    @ResponseBody
    @GetMapping("/paramquery2")
    public String getMethodQuery2(
       @RequestParam(name = "param", required = false, defaultValue = "Saludo opcional")
       String param) {
           String salida = html;
           salida=salida.replace("[CONTENIDO]", param);
           return salida;
    }
}
```

Otra posibilidad de pasar datos a un método es mediante el path variable. La URL seria:

http://dominio:puerto/target/path/pathVariable

Para recoger el valor Spring usa la anotación **@PathVariable** y completaremos el ath en **@GetMapping** con el nombre de la variable entre llaves.

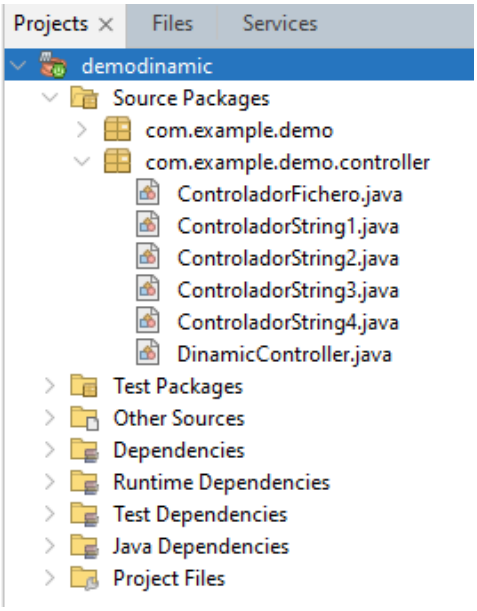
Para nuestro ejemplo con la URL:

http://localhost:8080/demodinamic/paramvariable/algo

DinamicController

```
@Controller
public class DinamicController {
    @ResponseBody
    @GetMapping("/paramvariable/{param}")
    public String getMethodParam(@PathVariable String param) {
        String salida = html;
        salida=salida.replace("[CONTENIDO]", param);
        return salida;
    }
}
```

La estructura del proyecto quedará:



Cuando queremos añadir a un controlador un Path extra en la URL y que todos los métodos incluidos en la clase lo usen, lo añadiremos con **@ResquestMapping**.

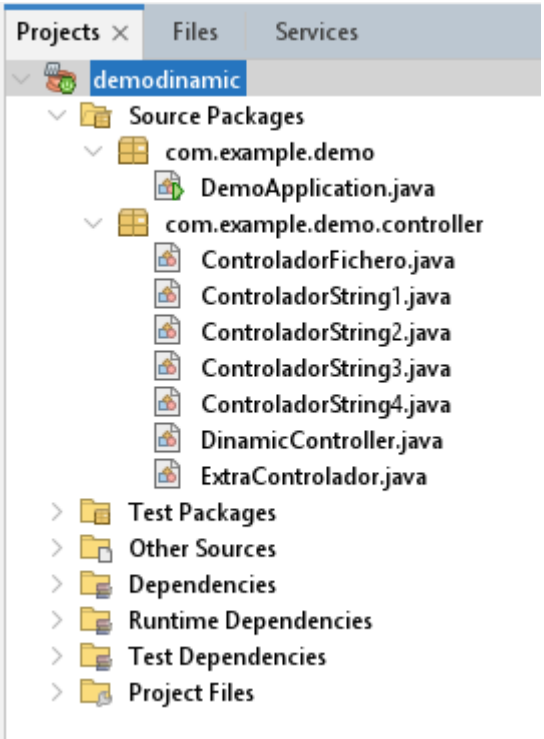
ExtraControlador

```
@Controller
@RequestMapping("/extra")
public class ExtraControlador {

    @ResponseBody
    @GetMapping("/nombre")
    public String getMethodParam() {
        return "Spring Dinamic";
    }

    @ResponseBody
    @GetMapping("/fecha")
    public String getMethodParam() {
        SimpleDateFormat dateFormat = new  SimpleDateFormat("yyyy");

        return dateFormat.format(new Date());
    }

}
```

Para nuestro ejemplo las dos URL serán:

http://localhost:8080/demodinamic/extra/nombre
http://localhost:8080/demodinamic/extra/fecha

Finalmente la estructura será:



5. Vista con Thymeleaf

En Spring, para mostrar resultados en HTML, podemos usar varios frameworks:

JSP+JSTL	-Tecnología antigua basada en servlets -Usa las marcas <% %> par embeber el código JSP
Moustache	-Motor de plantillas lógicamente limitado -Usa etiquetas simples ({{name}}, {{#list}})
Thymeleaf	-Motor de plantillas moderno -Usa atributos HTML (th:text, th:if, etc.) -Puede tratar datos en la vista (th:if, th:each, expresiones)

Aunque **Thymeleaf** es el más moderno y más usado de Spring Boot, en el entorno profesional de desarrollo en entorno web en la parte de **frontend** se usan más otros **frameworks** como **Angular**, **React** o **Vue** basados en tecnologías **Javascript**.

Podemos consultar la aplicación de cada una de estas tecnologías en:

- https://www.geeksforgeeks.org/html/list-of-front-end-technologies/

Y su uso en: https://trends.builtwith.com/

- https://trends.builtwith.com/framework/Angular
- https://trends.builtwith.com/javascript/React
- https://trends.builtwith.com/javascript/Vue

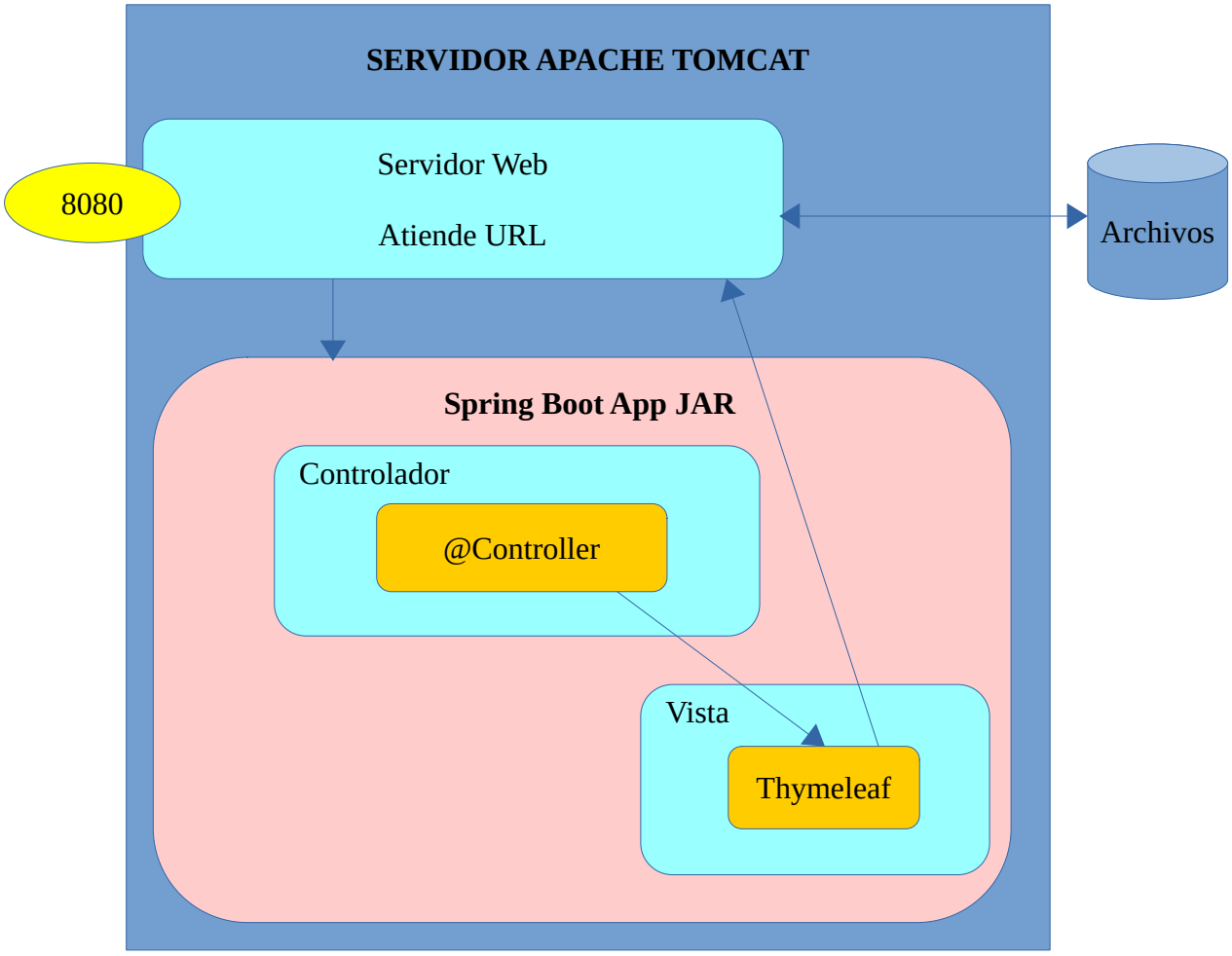
En nuestros próximos ejemplos usaremos **Thymeleaf** de forma básica para mostrar los datos procesados en Spring Boot mediante plantillas que recibirán a través de la clase **Model**.

Los controladores almacenarán los datos en **Model** y las plantillas de **Thymeleaf** podrán mostrarlos usando atributos de tipo **"th:…"**.

No se pretende crear aplicaciones para **frontend** con **Thymeleaf** sino mostrar y/o gestionar acciones básicas en entorno servidor.

El uso de Spring Boot en entornos profesionales se centra en gestionar a través de un API un acceso a datos, por ejemplo mediante **APIREST**. Nuestro objetivo final será no usar **Thymeleaf** en la VISTA sino devolver datos en formato **JSON** que sean tratados por aplicaciones del **frontend**.

El esquema de **Spring Boot** usando **Thymeleaf** sería:



5.1 Proyecto demo-thymeleaf-saludo

Vamos a crear un proyecto **demo-thymeleaf-saludo** con un controlador (como en demo-dinamic) que procese unos datos que recibe y redirija la salida a una plantilla de Thymeleaf que dinámicamente cree el contenido HTML con los datos procesados.

Al crearlo seleccionaremos las siguientes dependencias de Spring:

- **Spring Web**
- **Spring Boot Dev Tools**
- **Thymeleaf**

Como hemos comentado anteriormente, podemos recibir datos mediante **QueryString** o **PathVariable**.
En este proyecto vamos a crear varios métodos para atender varias URL.

La URL del proyecto es http://localhost:8080/demosaludo

La siguientes tablas muestran los métodos implementados:

Método	welcome
URL	http://localhost:8080/demosaludo
Path	/
Modelo	mensaje="¡Hola a todos!"
Plantilla Thymeleaf	index.html

SaludoController

```
@Controller
public class SaludoController {

    @GetMapping("/")
    public String welcome(Model model) {
        model.addAttribute("mensaje", "¡Hola a todos!");
        return "index";
    }
}
```

Con **Thymeleaf**, al hacer **return "index"** sin la anotación **@ResponseBody**, Spring buscará un archivo **index.html** en la carpeta **templates**

Cuando **index.html** recibe el **model** con los atributos, puede utilizar sus valores para mostrar datos como el de mensaje creado en el método **welcome** anterior:

```
<h1 th:text="${mensaje}">Mensaje</h1>
```

Usando @RequestParam:

Método	welcomeHola
URL	http://localhost:8080/demosaludo/hola**?name=xxxxxx**
Path	/hola
Modelo	saludo="Hola "+name
Plantilla Thymeleaf	hola.html

SaludoController

```
@Controller
public class SaludoController {
    @GetMapping("/hola")
    public String welcomeHola(@RequestParam("name") String name, Model model) {
        model.addAttribute("saludo", "Hola "+name);
        return "hola";
    }
}
```

Al usar **@RequestParam** podemos recoger datos en la parte **QueryString** de la URL de datos asignados al parámetro.

Cuando **hola.html** recibe el **model** con los atributos, puede utilizar sus valores para mostrar datos como el de mensaje creado en el método **welcome** anterior:

```
<h1 th:text="${saludo}">Mensaje de saludo</h1>
```

Usando @RequestParam con required y defaultValue

Método	welcomeHello
URL	http://localhost:8080/demosaludo/hello?**name=xxxxxx** http://localhost:8080/demosaludo/hello
Path	/hello
Modelo	saludo="Hola "+name
Plantilla Thymeleaf	hola.html

SaludoController

```
@Controller
public class SaludoController {
    @GetMapping("/hello")
    public String welcomeHello(
          @RequestParam(name="name", required=false, defaultValue="Student")
            String name,
          Model model) {
        model.addAttribute("saludo", "Hello "+name);
        return "hola";
    }
}
```

Al usar **@RequestParam** con **required=false, defaultValue="Student"** podemos recoger datos en la parte **QueryString** de la URL de datos asignados al parámetro, pero si no se ha definido le asignamos un valor por defecto.

Cuando **hola.html** recibe el **model** con los atributos, puede utilizar sus valores para mostrar datos como el de mensaje creado en el método **welcome** anterior:

```
<h1 th:text="${saludo}">Mensaje de saludo</h1>
```

Usando @PathVariable:

Método	welcomeHolaPath
URL	http://localhost:8080/demosaludo/hola/**xxxx**
Path	/hola/{name}
Modelo	saludo="Hola "+name
Plantilla Thymeleaf	hola.html

SaludoController

```
@Controller
public class DinamicController {
    @GetMapping("/hola/{name}")
    public String welcomeHolaPath(@PathVariable String name, Model model) {
        model.addAttribute("saludo", "Hola "+name);
        return "hola";
    }
}
```

Al usar **@PathVariable** podemos recoger datos en la parte variable de la URL de datos extra.
En este ejemplo es lo añadido después de **/hola/**

Recuerda que los archivos de plantilla de Thymeleaf, a diferencia de un HTML habitual, comienzan por:

Platilla .html de Thymeleaf

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   ...
```

5.2 Proyecto demo-calculadora

Se desea desarrollar un proyecto denominado **demo-calculadora** de **Spring Boot** usando **Thymeleaf** que atienda un método GET con el **PathVariable**

`/suma/{op1}/{op2}`

El método recibe **op1** y **op2** que deben ser dos números enteros y devolverá la suma de ellos en una variable del modelo denominada **resultado**.

Ejemplo:
Con la URL http://localhost:8080/democalculadora/suma/34/15
el mensaje a mostrar será:

Suma 34+15 = 49

En la plantilla de **Thymeleaf** para mostrar el mensaje se deberá incluir:

Calculadora.html

```
...

  <div class="container">
     <h1>CALCULADORA</h1>
     <p th:text="${resultado}">Resultado de la operación aritmética</p>
  </div>

...
```

Además, se incluye un **index.html** en static que muestra enlaces de ejemplo a los métodos implementados.

Las anotaciones necesarias para desarrollar el controlador del proyecto son:

- @Controller
- @GetMapping
- @PathVariable

5.3 Proyecto demo-thymeleaf-objects

En este proyecto se pretende mostrar algunos de los **objetos de utilidad** disponibles en **Thymeleaf**. Estos objetos/variables especiales nos proporcionan métodos para manipular diferentes tipos de datos.

Los más importantes son:

#temporals	para manipular **fechas/horas**
#strings	para manipular **cadenas/strings**
#numbers	para formatear la salida de **números**
#objects	para manipular **objetos**
#bools	para manipular **booleans**
#arrays	para manipular **arrays**
#lists	para manipular **lists/listas**
#locale	para obtener información de locale (país, lenguaje, ...)
session **context** **request**	son objetos con información de contexto, la sesión y la URL

En este proyecto el controlador **EjemploControlador** cargará diferentes tipos de datos en el **model** como strings, fechas, números, arrays, listas, objetos, … que serán manipulados por **Thymeleaf** con sus objetos de utilidad en el archivo de la vista **ejemplo-completo.html**

Ejemplos de URL:

```
http://localhost:8080/demothymeleafobjects
http://localhost:8080/demothymeleafobjects/ejemplo
```

5.4 Proyecto demo-today

Ejemplo básico para mostrar datos del día actual.

Desde el controlador **DateController** cargaremos en el **model** datos obtenidos con Java y luego ya en la vista **current-date.html** mostraremos esos datos y otros obtenidos con **#temporals** en **Thymeleaf**.

Ejemplos de URL:

```
http://localhost:8080/demotoday
http://localhost:8080/demotoday/today
```

5.5 Proyecto demo-thymeleaf-bucles

En este ejemplo se muestra el uso de **th:each** en **Thymealeaf** par mostrar **arrays**, **listas** o **matrices**.

5.6 Proyecto demo-thymeleaf-model

Para poder almacenar datos solemos usar objetos que se instancian de clases de datos. Todas las clases de datos creadas se definen en el Java Package model.

En este ejemplo definimos una clase de datos **User** con tres propiedades:

User

```
public class User implements Serializable {

    private String name;
    private String email;
    private int age;
```

Las clases de datos deben tener al menos:

- Constructor vacío
- Constructor completo
- Getters y Setters
- toString()

En el controlador podemos añadir un método para predefinir atributos del **model**.
En nuestro proyecto, en el controlador **UserController** se ha creado un atributo **users** con una lista de 3 usuarios de ejemplo usando **@ModelAtribute**.

Los métodos, pueden gestionar esta lista para devolver la lista existente de todos los usuarios o un solo usuario.

Ejemplos de URL:

http://localhost:8080/demothymeleaf
http://localhost:8080/demothymeleaf/users
http://localhost:8080/demothymeleaf/details

6. Arquitectura MVC en SPRING

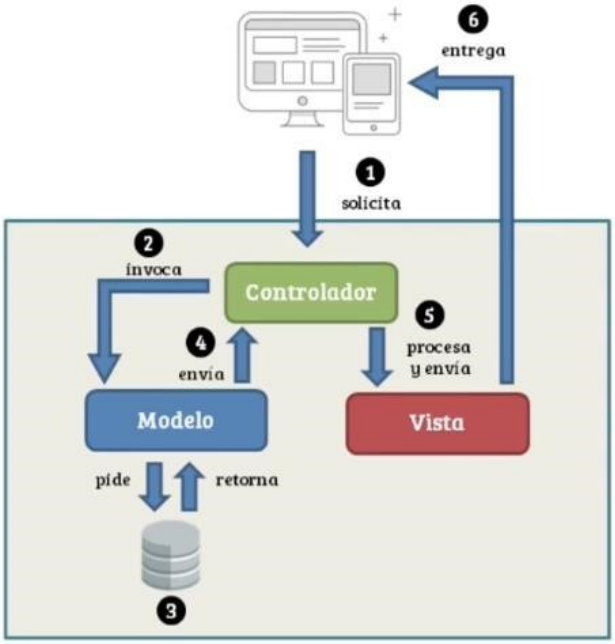
El **Modelo Vista Controlador (MVC)** es un patrón de diseño arquitectónico que separa la representación de la información de la interacción del usuario, facilitando la gestión y escalabilidad de aplicaciones complejas. Este patrón se utiliza frecuentemente en el desarrollo de software, especialmente en aplicaciones web y desarrollo de interfaces gráficas.

Componentes del MVC

- **Modelo**:
  - Representa la lógica de datos y la lógica de negocio de la aplicación.
  - Se encarga de gestionar la información, incluyendo su recuperación, almacenamiento y procesamiento.
  - No tiene conocimiento directo de la vista, lo que permite que la lógica de negocio esté desacoplada de la presentación.

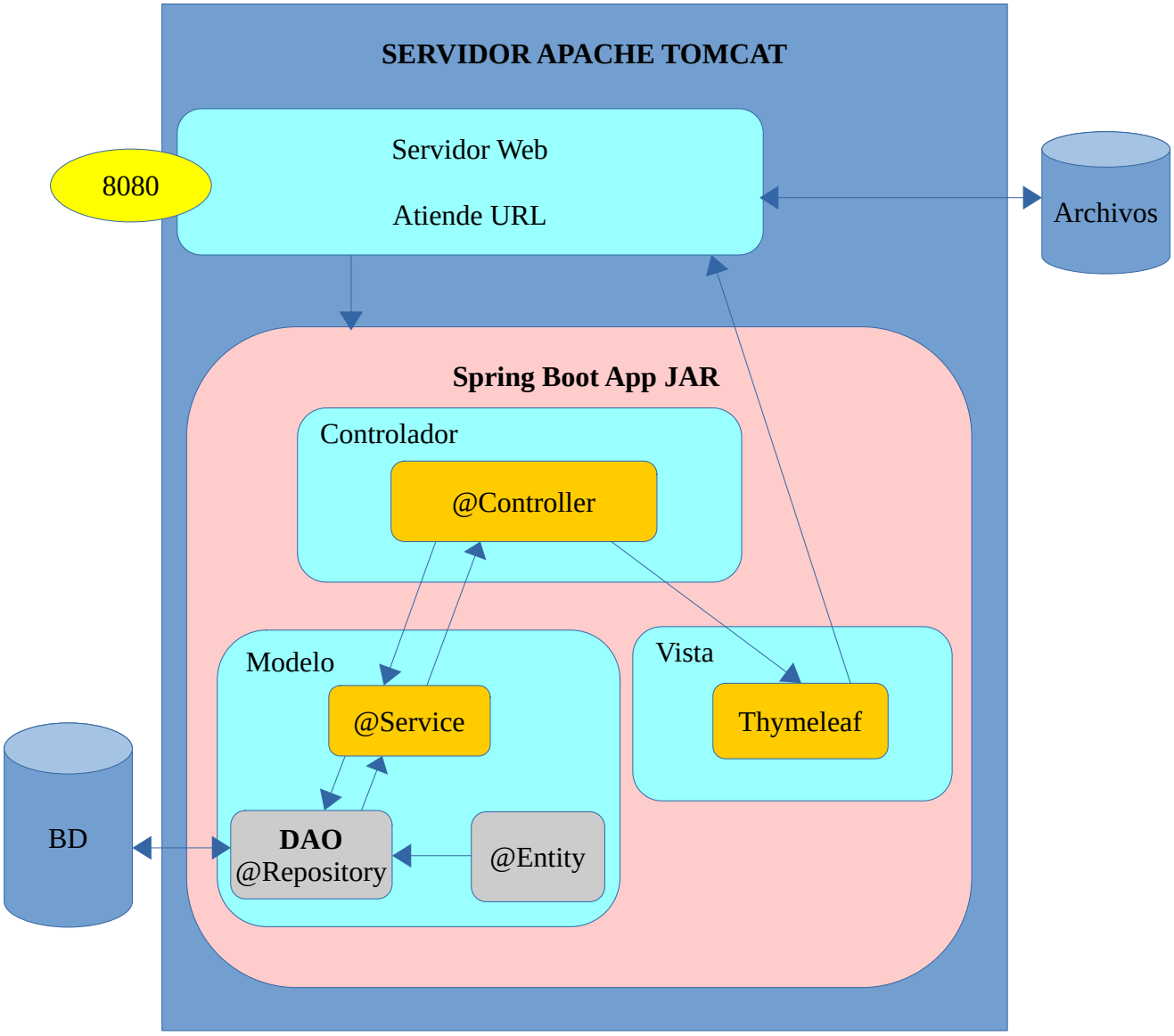
- **Vista**:
  - Se ocupa de la representación visual de los datos, presentando la información que el usuario necesita.
  - Escucha eventos generados por el controlador y actualiza la interfaz de usuario en consecuencia.
  - Puede estar compuesta por múltiples componentes que permiten mostrar los datos de manera amigable y accesible.
  - Utiliza plantillas (normalmente HTML) que tras ser procesadas, serán visualizadas por el navegador junto con otros recursos estáticos (CSS, imágenes, JS, etc,).

- **Controlador**:
  - Actúa como intermediario entre el modelo y la vista.
  - Recibe las interacciones del usuario desde la vista y las traduce en acciones que afectan al modelo.
  - Actualiza la vista con los resultados o cambios en el modelo después de que se procesan las acciones del usuario.



Para completar el MVC utilizando Spring añadiremos un nuevo componente que identificaremos con **@Service** y que tendrá los métodos que realizarán los procesos necesarios para obtener la información solicitada.

En la mayoría de casos se necesitarán clases que accedan a las BD (Bases de Datos) como las **@Repository** y clases que almacenen los datos mapeados en memoria que serán los **@Entity**.



Vamos a crear un proyecto que

6.1 Proyecto demo-thymeleaf-service

Este proyecto es un ejemplo básico pero completo de una aplicación web con **Spring Boot MVC y Thymeleaf**, mostrando el patrón **Modelo-Vista-Controlador** en acción.

Dada una carga de datos de usuarios definida en la clase **UserService**, se desea mostrar la lista de usuarios usando la plantilla **users.html** y mostrar los detalles de uno de ellos mediante la plantilla **details.html**.

El flujo del **MVC** permitirá atender desde el controlador **UserController** las URL de los **Endpoints,** que cargará el **model** con los datos obtenidos a través de **UserService** para las plantillas de **Thymeleaf** que crea la **Vista**.

Los **Endpoints** funcionales son:

	Controlador	Vista
`GET /`	-	index.html
`GET /users`	showUsers()	users.html
`GET /details/{id}`	detailsUser()	details.html

Características **Thymeleaf** utilizadas:

- **th:each**: Iteración sobre colecciones (users)
- **th:text**: Mostrar valores (${user.name})
- **th:href**: Enlaces dinámicos (@{/details/{id}(id=${user.id})})
- **th:if**: Condicionales (para mensajes de error)
- Expresiones: ${users}, ${user.id}, etc.

La anotación **@Service** define el componente de servicio y
@Autowired en el controlador nos permite inyectar la dependencia del servicio.

Para comprenderlo mejor, es equivalente:

`Con @Autowired`	`@Controller`
	`public class UserController {`
	`  `**`@Autowired`**
	`  `**`private UserService userService;`**
	`  ...`

`GET /users`	`@Controller`
	`public class UserController {`
	`  `**`private UserService userService;`**
	
	`  `**`public UserController(UserService userService) {`**
	`  `**`this.userService = userService;`**
	`  `**`}`**
	`  ...`

La diferencia entre las distintas versiones del proyecto está en la carga de los datos de los usuarios.

Demo-thymeleaf-service-v1

Partiendo del proyecto **demo-thymeleaf-model** hemos creado un componente de tipo **@Service** denominado **UserService** que almacenará la lista de usuarios (en vez de usar el **@ModelAtribute**) e implementa dos métodos para obtener toda la lista (**findAll**) o solo un usuario (**findById**).

El controlador **UserController** accederá al **UserService** para obtener los datos y los pasará a la vista a través del **model**.

Se ha cambiado en la vista de **users.html** los enlaces en cada registro para ver el detalle de ese usuario.

> **Ejemplos de URL**:
> http://localhost:8080/demoservice1
> http://localhost:8080/demoservice1/users
> http://localhost:8080/demoservice1/details/1

Demo-thymeleaf-service-v2

Partiendo del proyecto **demo-thymeleaf-service-v1** cambiamos el **UserService** para cargar los datos usando el método nuestro private **loadInitialUsers** en el constructor.

> **Ejemplos de URL**:
> http://localhost:8080/demoservice2
> http://localhost:8080/demoservice2/users
> http://localhost:8080/demoservice1/details/1

Demo-thymeleaf-service-v3

Partiendo del proyecto **demo-thymeleaf-service-v1** cambiamos el **UserService** para cargar los datos usando un método con la anotación **@PostConstruct**

> **Ejemplos de URL**:
> http://localhost:8080/demoservice3
> http://localhost:8080/demoservice3/users
> http://localhost:8080/demoservice3/details/1

Los componentes **@Repository** y **@Entity** los veremos en siguiente apartado con **SPRING DATA** donde incorporaremos el acceso a bases de datos.

6.2 Proyecto demo-thymeleaf-bingo-service

Como repaso y aplicación práctica de todo lo aprendido crearemos este proyecto de una aplicación web con **Spring Boot MVC y Thymeleaf**, mostrando el patrón **Modelo-Vista-Controlador** en acción.

Deseamos generar con Spring Boot un juego de bingo.

Al comenzar se mostrará el contenido de **reset.html** que permitirá comenzar el bingo con **75 o 90 bolas** mediante dos botones.

Al comenzar el juego se mostrará el contenido de **bingo.html** que contendrá el siguiente contenido dinámico generado en el controlador y almacenado en el **model**:

- **Número**: es el último número generado aleatoriamente
- **Rango**: usando el valor de PathVariable (75 o 90) recibido al pulsar el botón mostrará el texto recibido "1-75" o "1-90"
- **Hora**: mostrará la hora y minutos actual
- **Restantes**: valor de las bolas que quedan por generar
- **Historial**: ArrayList con las bolas generadas hasta el momento inversamente ordenadas. Para ello se ha creado una clase Bola con los valores a mostrar en cada card del historial.

Bola.java

```
public class Bola implements Serializable {
    Integer numero;
    String hora;
    Integer orden;
    String grupo;
    String clase;
      ...
}
```

El controlador atenderá las siguientes URL

/juegos/bingo	redirige a /juegos/bingo/reset
/juegos/bingo/reset	reinicia el juego vaciando el historial y muestra **reset.html**
/juegos/bingo/{max}	genera un número, crea la bola y la almacena en el historial, graba la información en el model y muestra **bingo.html**

En **Thymeleaf** se ha usado:

- **th:href**="@{/css/style.css}"
- **th:class**="${btn_class}"
- **th:href**="${btn_href}"
- **th:class**="${numero <= 15 ? 'ball-circle pulse history-ball-15' :
  numero <= 30 ? 'ball-circle pulse history-ball-30' :
  numero <= 45 ? 'ball-circle pulse history-ball-45' :
  numero <= 60 ? 'ball-circle pulse history-ball-60' :
  numero <= 75 ? 'ball-circle pulse history-ball-75' :
  'ball-circle pulse history-ball-90'}"
- **th:text**="${rango}"
- **th:text**="${hora}"
- **th:text**="${resto}"
- **th:each**="bola : ${historial}
- **th:text**="${bola.orden}"
- **th:class**="${bola.clase}"
- **th:text**="${bola.numero}"
- **th:text**="${bola.grupo}"
- **th:text**="${bola.hora}"