SPRING

Spring es un framework de desarrollo cuya base es el lenguaje Java, aunque también ofrece soporte para Kotlin y, de manera limitada, para Groovy.
Es uno de los frameworks más populares y ampliamente utilizados para crear aplicaciones Java modernas, gracias a su modularidad, facilidad para gestionar dependencias y amplio ecosistema de proyectos complementarios.

Spring es un framework para Java que ayuda a:

- Gestionar objetos y dependencias automáticamente.
- Separar responsabilidades entre las capas de tu aplicación (modelo, vista, controlador).
- Facilitar la programación modular y la prueba de código.
- Crear aplicaciones web, microservicios y sistemas complejos con menos código repetitivo.

Su núcleo se basa en la idea de Inversión de Control (IoC) y Programación Orientada a Aspectos (AOP).

Spring se ha expandido en múltiples módulos especializados:

- **Spring MVC:** para aplicaciones web.
- **Spring Data:** para bases de datos.
- **Spring Security:** para autenticación y autorización.
- **Spring Cloud:** para microservicios.
- **Spring Boot:** para simplificar la configuración y despliegue.

Inversión de Control (IoC) y Dependencia

Uno de los conceptos más importantes de Spring.

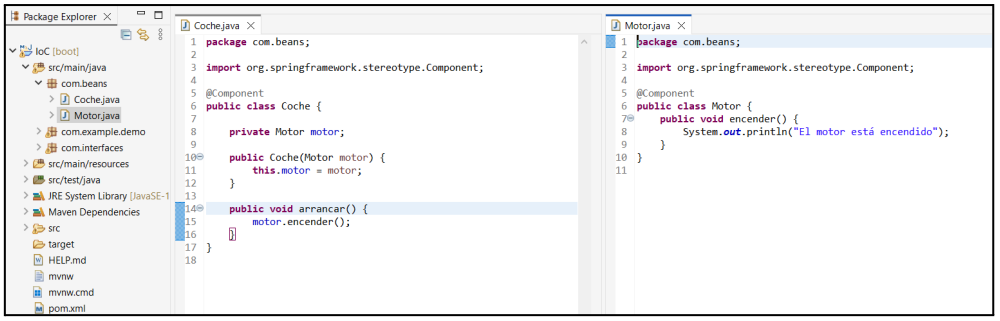
- IoC significa que el framework se encarga de crear y administrar los objetos de tu aplicación, en lugar de que tú lo hagas manualmente.
- En lugar de instanciar objetos con new, Spring los crea, configura y conecta automáticamente según las anotaciones y configuraciones definidas.
- Esto se logra mediante el contenedor de Spring, que gestiona los beans (objetos gestionados por Spring).
- Resuelve las dependencias entre beans automáticamente.
- Permite cambiar implementaciones fácilmente, usando interfaces o configuraciones externas.

Spring tiene dos tipos de contenedores IoC

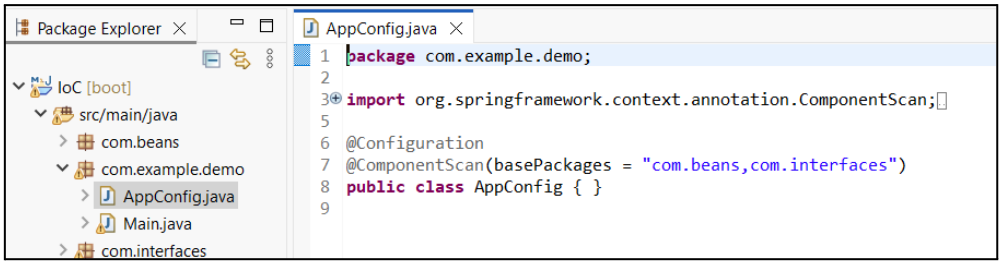
- BeanFactory: Contenedor básico, carga beans bajo demanda.
- ApplicationContext: Más avanzado, carga todos los beans al inicio, soporta eventos, internacionalización y más. Es el más usado.

Ejemplo

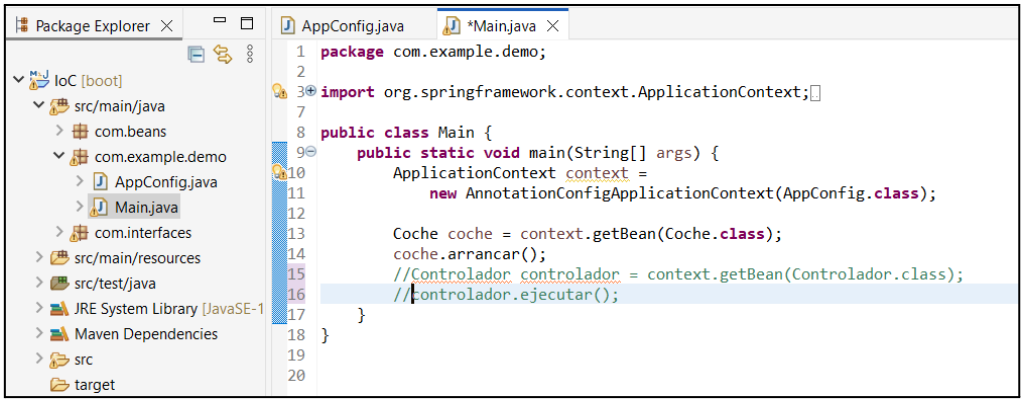
En estas clases declaramos los Beans(Componentes) que se cargan al iniciar el programa

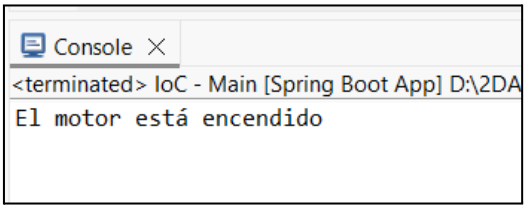


Aquí estamos declarando una clase con la configuración y esta clase sabrá donde se ubican los componentes que vamos a utilizar mediante el componentScan



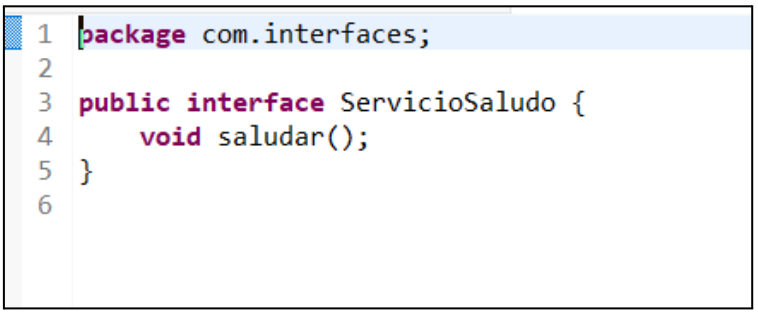
Aquí estamos declarando el ApplicationContext para poder crear la clase con la que puede leer las clases que leen las anotaciones con ella podremos gestionar los beans de la aplicación





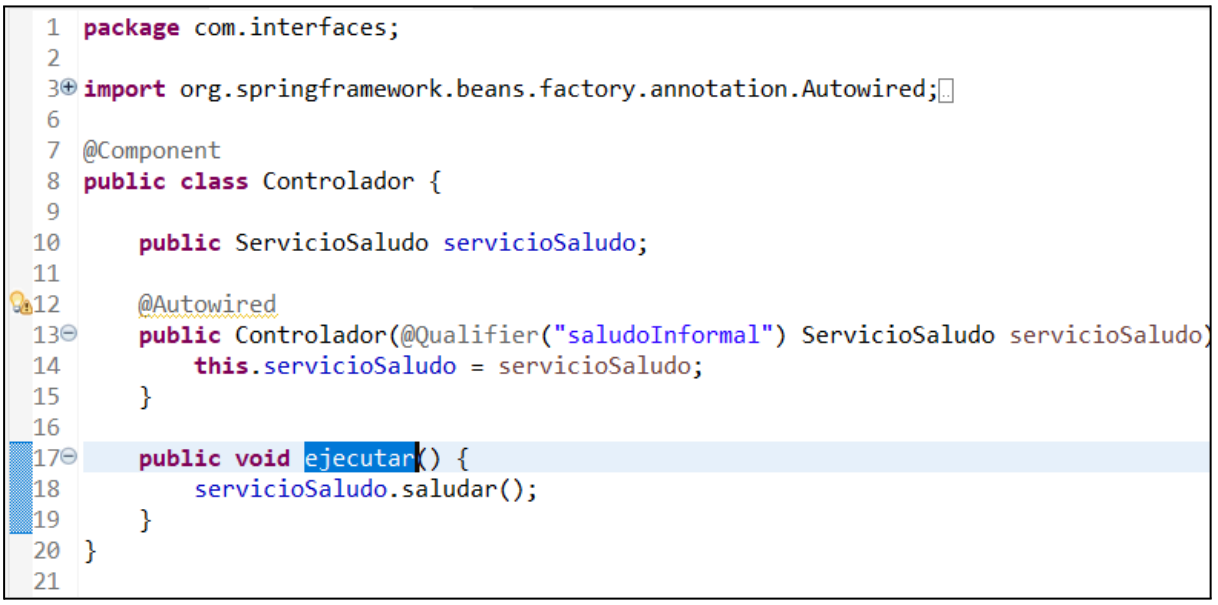
Interfaces

Vamos a proceder a crear una interfaz que sirve como modelo.

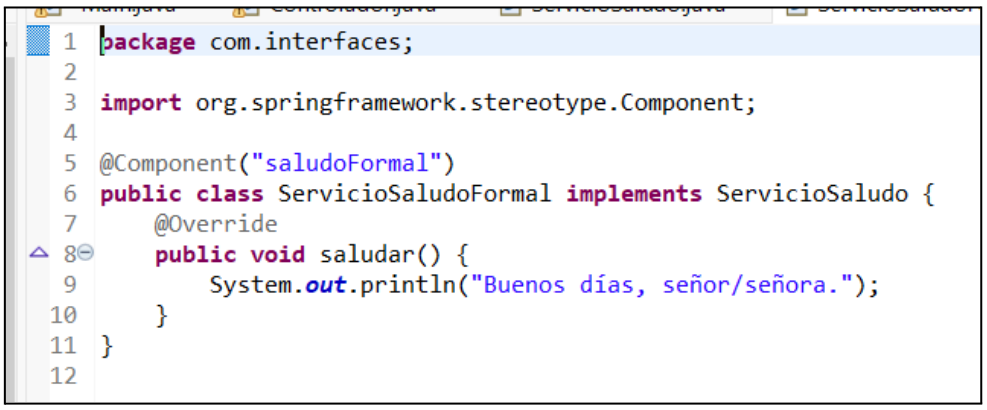


```java
package com.interfaces;

public interface ServicioSaludo {
    void saludar();
}
```

Vamos a crear el componente con un condicionante para elegir que tipo de saludo queremos



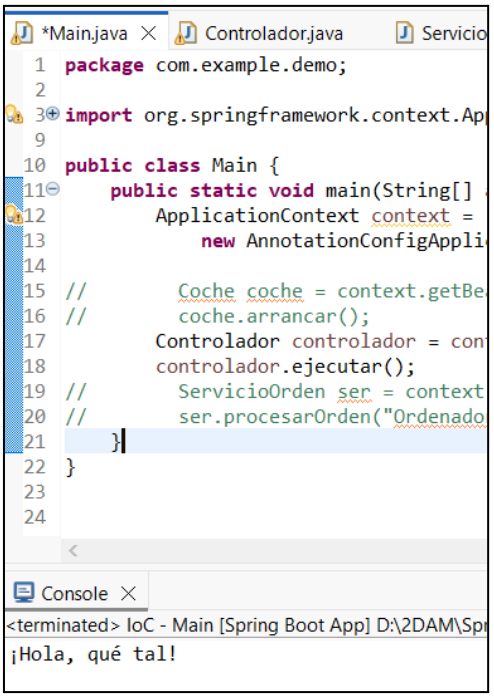
```java
package com.interfaces;

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class Controlador {

    public ServicioSaludo servicioSaludo;

    @Autowired
    public Controlador(@Qualifier("saludoInformal") ServicioSaludo servicioSaludo)
        this.servicioSaludo = servicioSaludo;
    }

    public void ejecutar() {
        servicioSaludo.saludar();
    }
}
```



```java
package com.interfaces;

import org.springframework.stereotype.Component;

@Component("saludoFormal")
public class ServicioSaludoFormal implements ServicioSaludo {
    @Override
    public void saludar() {
        System.out.println("Buenos días, señor/señora.");
    }
}
```

```java
package com.interfaces;

import org.springframework.stereotype.Component;

@Component("saludoInformal")
public class ServicioSaludoInformal implements ServicioSaludo {
    @Override
    public void saludar() {
        System.out.println("¡Hola, qué tal!");
    }
}

```

Aquí vemos el código principal para poder ejecutar la interfaz



Programación Orientada a Aspectos

La AOP es un paradigma de programación que permite separar las funcionalidades transversales de la lógica principal de tu aplicación.

Son tareas que afectan a varias partes del código, pero no forman parte de la lógica principal, como:

- Logging (registro de eventos)
- Seguridad (autenticación/autorización)
- Manejo de transacciones
- Caché
- Métricas o monitoreo

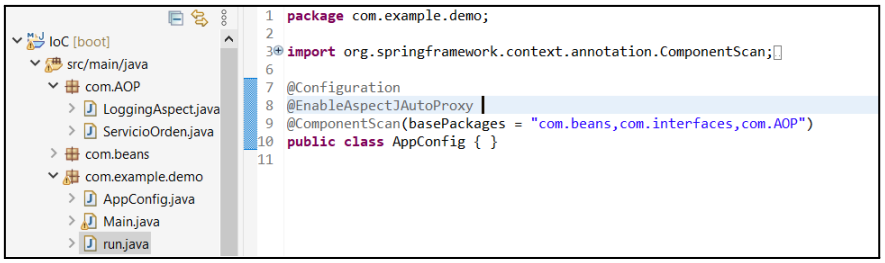
  Separa esas funcionalidades en aspectos, y el framework las “inyecta” donde se necesiten, sin tocar la lógica principal.

Ejemplo

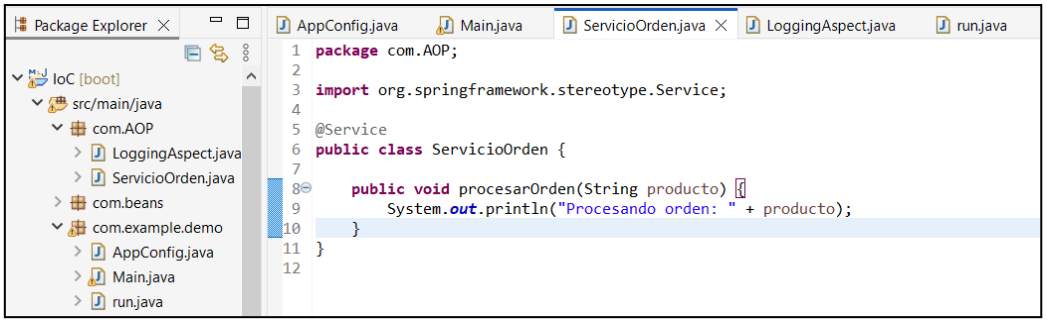
Necesitamos añadir esta dependencias para empezar a trabajar con esta parte de Spring

```xml
<!-- Dependencias de AOP -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Deberemos añadir esa anotación en el App Config si vamos a seguir ejecutandolo como siempre



Luego deberemos crear el servicio que queremos que maneje el log que crearemos con la anotación @Service para mencionar que es la lógica del servicio



Luego procederemos a crear la clase para que reaccione antes de que se ejecute el código en la que pondremos @Aspect que nos indica que tiene codigo transversal lo que nos permite no duplicar código pero que afecte a varios puntos del código

```java
package com.AOP;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.AOP.ServicioOrden.procesarOrden(..))")
    public void logAntesDeProcesar() {
        System.out.println("LOG: Antes de procesar la orden");
    }
}
```

Y por último el Main en el que veremos como se ejecuta pero existe una forma de que podamos ver de que ya no tengamos que usar app Config para que gestione la configuración.

```java
package com.example.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.AOP.*;
import com.beans.*;
import com.interfaces.Controlador;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfig.class);

//        Coche coche = context.getBean(Coche.class);
//        coche.arrancar();
//        Controlador controlador = context.getBean(Controlador.class);
//        controlador.ejecutar();
        ServicioOrden ser = context.getBean(ServicioOrden.class);
        ser.procesarOrden("Ordenador");
    }
}
```

De esta forma estamos ejecutando la aplicación de Spring por primera vez y al decirle a la aplicación lo que debe tener en cuenta para trabajar el AppConfig ya no necesita estar ni tendremos que crear un objeto de AnnotationConfigApplicationContext ya que al correr la aplicación ya está generando un ApplicationContext

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.AOP.ServicioOrden;

@SpringBootApplication(scanBasePackages = {"com.AOP", "com.beans", "com.interfaces"})
public class run {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(run.class, args);

        ServicioOrden ser = context.getBean(ServicioOrden.class);
        ser.procesarOrden("Ordenador");
    }
}
```

```
Console
<terminated> IoC - Main [Spring Boot App] D:\2DAM\SpringR
LOG: Antes de procesar la orden
Procesando orden: Ordenador
```

Anotaciones:

- @Autowired: Permite inyectar automáticamente dependencias (beans) dentro de una clase, sin necesidad de crear los objetos manualmente.
- @Configuration: Indica que la clase contiene la configuración de Spring (equivale a un archivo XML antiguo). Dentro de ella se pueden declarar beans con la anotación @Bean.
- @Component: Marca una clase como un componente genérico que será detectado automáticamente por Spring y gestionado como un bean.
- @ComponentScan: Indica a Spring qué paquetes debe escanear para encontrar clases anotadas con @Component, @Service, @Controller, etc.
- @Service: Sirve para mencionar que es la lógica del servicio, es decir, la capa de negocio de la aplicación.
- @Repository: Marca una clase como parte de la capa de acceso a datos (DAO o repositorio). También traduce excepciones de base de datos a excepciones propias de Spring.
- @Aspect: Nos indica que tiene código transversal, lo que nos permite no duplicar código pero que afecte a varios puntos del programa (por ejemplo logs o seguridad).
- @Controller: Anota la clase como un controlador de Spring MVC, encargado de manejar las peticiones web y devolver vistas (HTML, JSP, etc.).
- @GetMapping("/"): Anotación que mapea la URL raíz del sitio web ("/") a este método de controlador. También existen @PostMapping, @PutMapping, @DeleteMapping, etc.
- @RestController: Se utiliza para crear controladores en Spring Boot que devuelvan datos serializados (normalmente en formato JSON) por defecto, sin usar vistas.

Spring Boot:

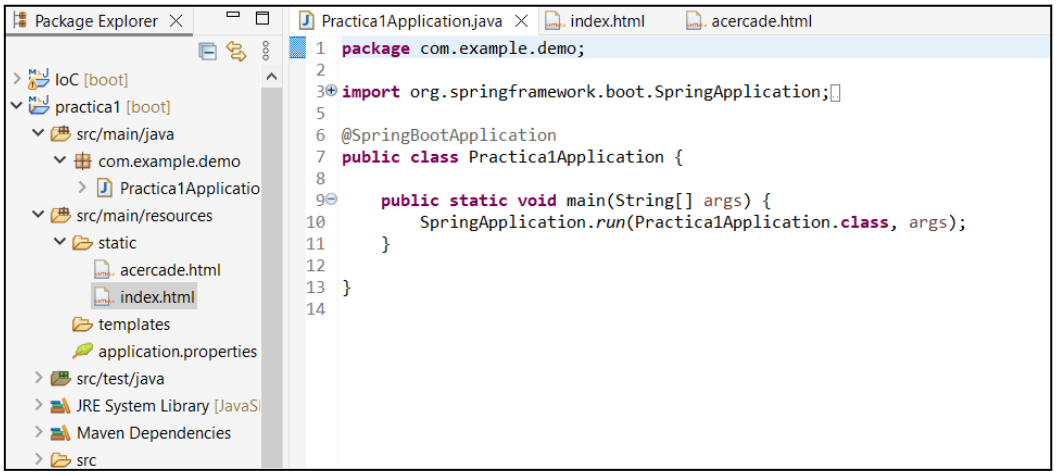
Spring Boot nace como una evolución de Spring que busca hacerlo rápido, simple y automático. Tiene como objetivo la creación de listas para poder ejecutar con la mejor configuración posible.

Dispone de su propio servidor embebido que es Apache Tomcat por defecto.

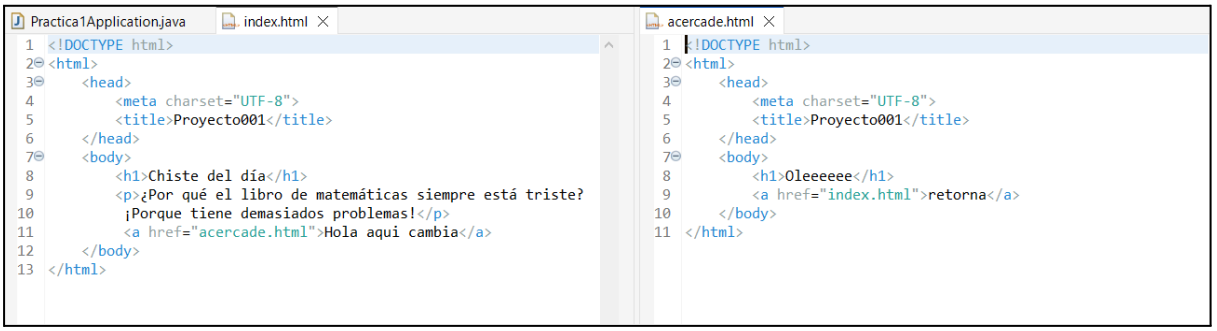
Yo voy a explicar Spring Boot centrándose más en la parte web. Tenemos la opción de crear páginas web estáticas y también disponemos de la creación de páginas dinámicas.

Ejemplo de pagina web estatica:

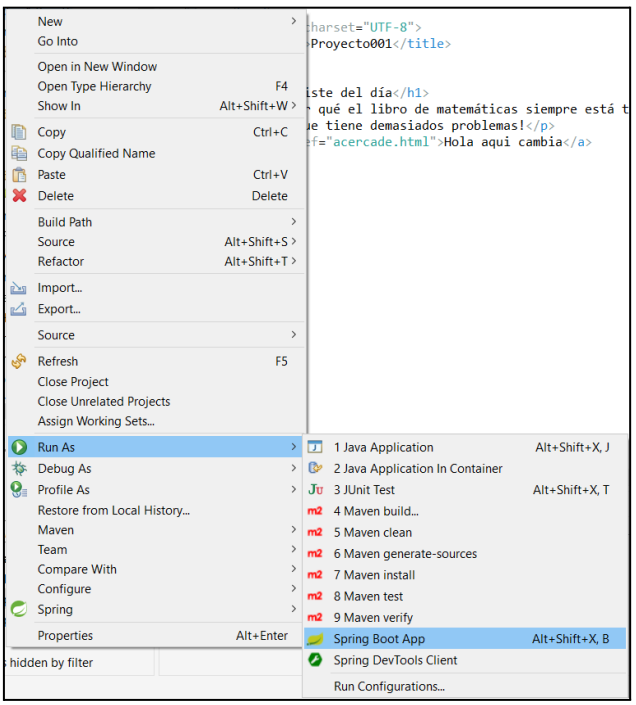
Cuando vayamos a crear el proyecto de Spring trabajaremos con maven entonces deberemos asignar esa propiedad en el typo y cuando nos pida las dependencias deberemos añadir la de Spring web



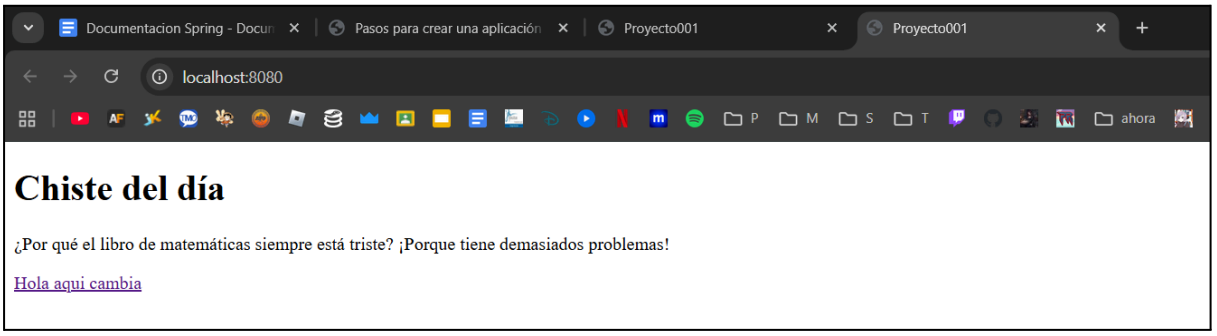
Como vamos a ver páginas estáticas pondremos por ahora el index y nuestro otra página html para hacer la prueba



Luego para poder verlo deberemos darle click derecho al proyecto y ejecutarlo como spring boot application.



Al darle estaremos iniciando el servidor que por defecto iniciara el index en http://localhost:8080/



Ejemplo de página web dinámica:

Para poder hacer que una página web sea dinámica deberemos realizar los mismos pasos de creación de un proyecto que el anterior con la diferencia de que deberemos añadir una dependencia llamada Thymeleaf.

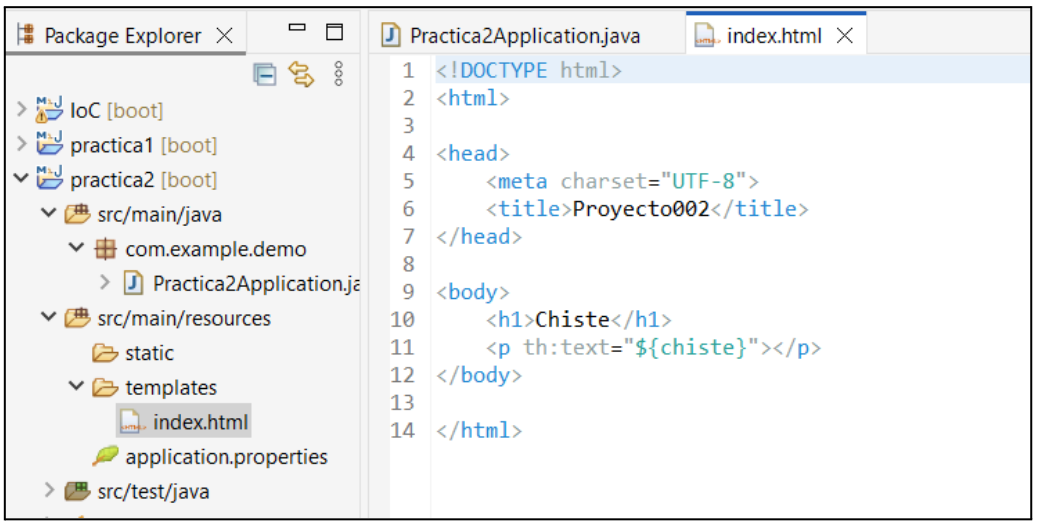
Actualmente la forma en la que vamos a trabajar será una que luego se cambiara pero para demostrar la forma más cómoda de una página dinámica será la siguiente

Practica2Application.java — index.html

```java
package com.example.demo;

import java.util.ArrayList;

@Controller
@SpringBootApplication
public class Practica2Application {

    public static void main(String[] args) {
        SpringApplication.run(Practica2Application.class, args);
    }

    @GetMapping("/")
    public String inicio(Model model) {
        List<String> chistes=new ArrayList<>();
        chistes.add("¿Qué le dice un 0 a un 8? Bonito cinturón.");
        chistes.add("¿Qué hace una abeja en el gimnasio? ¡Zum-ba!");
        chistes.add("¿Cuál es el colmo de un electricista? Que su hijo sea un apagado.");
        chistes.add("¿Por qué estás hablando con la pared? ¡Porque la mesa no me responde!");

        String chisteAzar = chistes.get((int)(Math.random()*chistes.size()));

        model.addAttribute("chiste", chisteAzar);
        return "index";
    }
}
```

Ahora deberemos crear el archivo html en la carpeta template en vez de static:

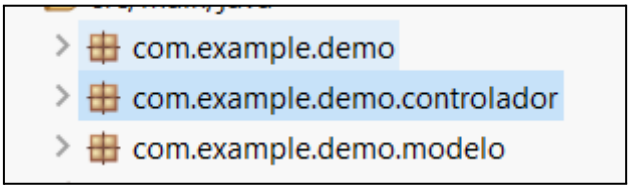
La etiqueta th:text="${chiste}" es una expresión de Thymeleaf que nos permite darle atributos a elementos del html para poder vincularlo al modelo de la aplicación a la vista y lo que se encuentra entre corchetes es nombre del atributo para luego poder referenciar.



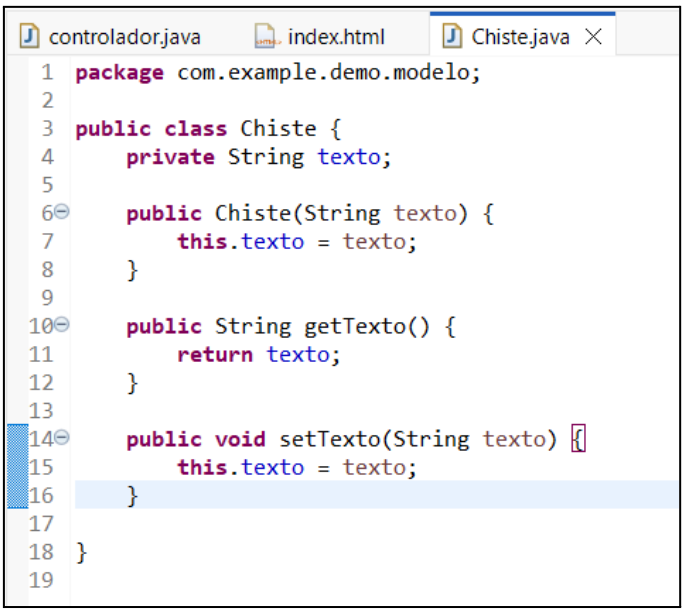
Si ejecutamos la página podremos ver que si la recargamos irá cambiando el chiste que se ve reflejado

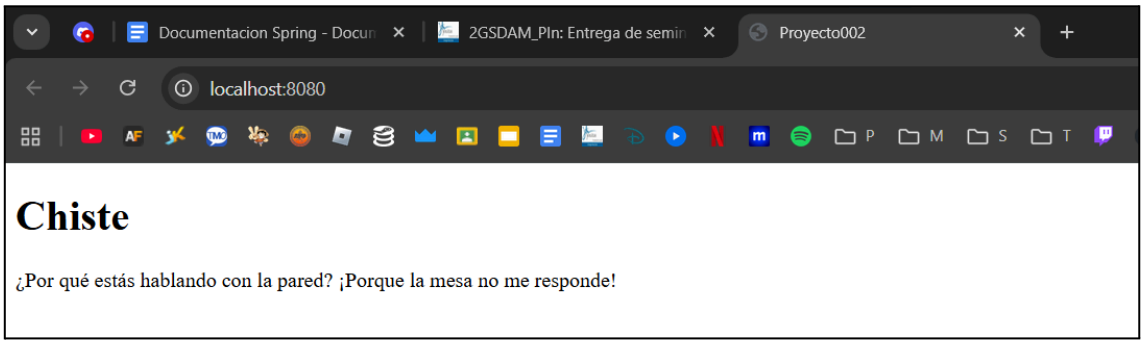
Ahora vamos a realizar nuevamente el mismo ejemplo anterior pero vamos a acomodarlo al patrón MVC

Vamos a crear un nuevo proyecto donde podremos ver la diferencia con el anterior y deberemos crear las siguientes carpetas que deben ir ubicadas dentro de la carpeta donde se ejecutará el main



En el modelo deberemos añadir la clase





Luego en el controlador deberemos hacer lo siguiente:

```java
package com.example.demo.controlador;

import java.util.ArrayList;

@Controller
public class controlador {

    @GetMapping("/")
    public String inicio(Model model) {
        List<Chiste> chistes = new ArrayList<>();
        chistes.add(new Chiste("¿Qué le dice un 0 a un 8? Bonito cinturón."));
        chistes.add(new Chiste("¿Qué hace una abeja en el gimnasio? ¡Zum-ba!"));
        chistes.add(new Chiste("¿Cuál es el colmo de un electricista? Que su hijo sea un apagado."));
        chistes.add(new Chiste("¿Por qué estás hablando con la pared? ¡Porque la mesa no me responde!"));

        Chiste chisteAzar = chistes.get((int)(Math.random() * chistes.size()));

        model.addAttribute("chiste", chisteAzar);
        return "index";
    }
}

```

Luego el html será prácticamente igual con la pequeña diferencia que donde está la expresión de chiste deberemos cambiarla a chiste.texto.
Esto hará que haga un getter a la clase de chiste para recibir el valor de texto

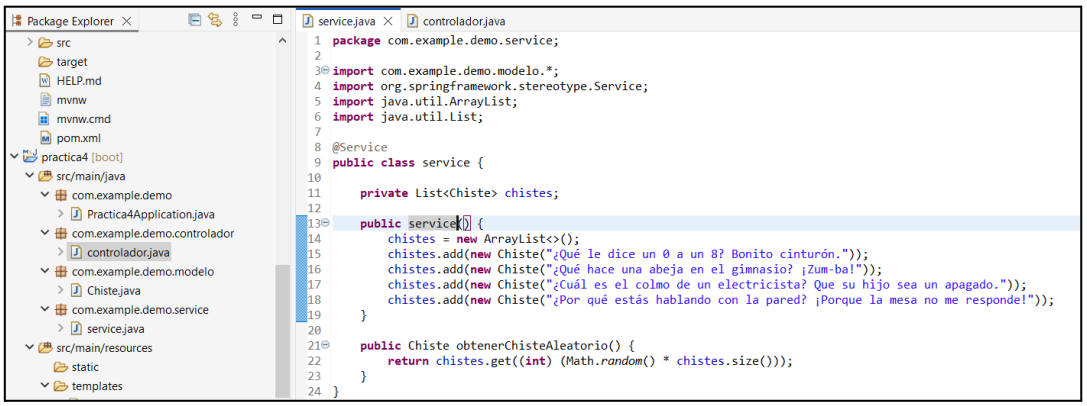
controlador.java | index.html | Chiste.java

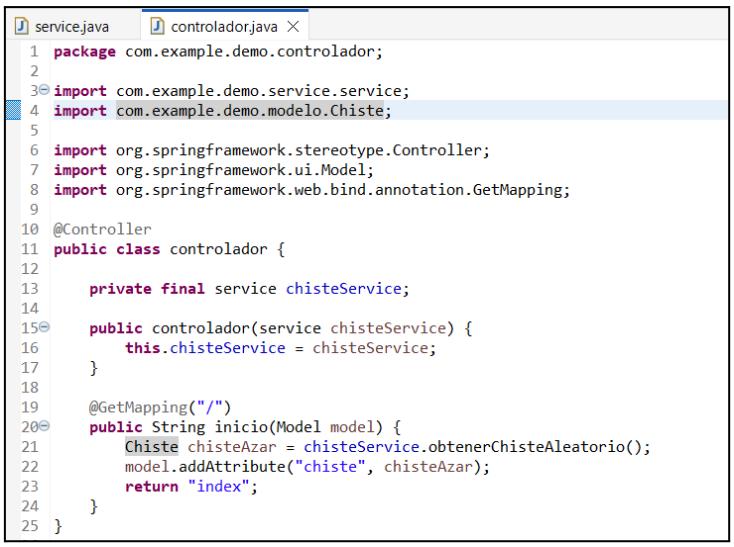
```html
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8">
    <title>Proyecto002</title>
</head>

<body>
    <h1>Chiste</h1>
    <p th:text="${chiste.texto}"></p>
</body>

</html>
```

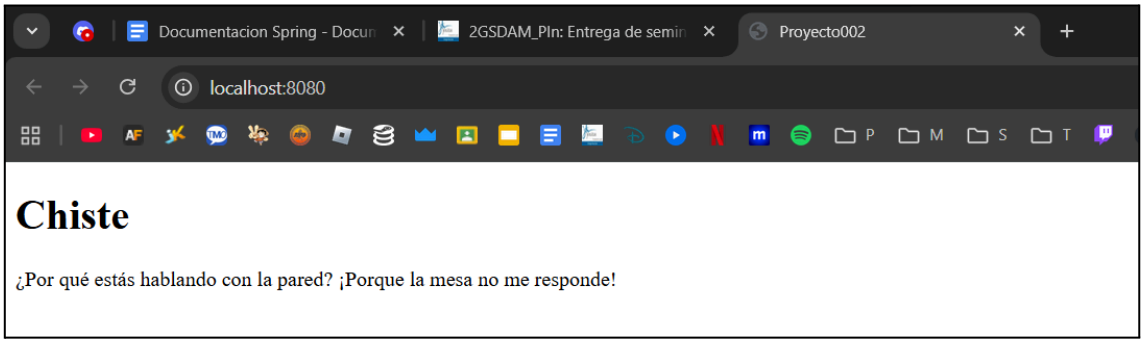
Ahora vamos a proceder a asignar a que los chistes se gestionen por servicio.

Podemos partir del mismo proyecto haciendo unos cambios:

Añadiremos un package donde estará ubicada el servicio con el siguiente código haciéndonos cambiar el controlador







Ahora vamos a proceder a hacer que sea una API:

Deberemos crear un nuevo proyecto sin la necesidad del Thymeleaf ya que solo vamos a comprobar que los datos se pasan correctamente.

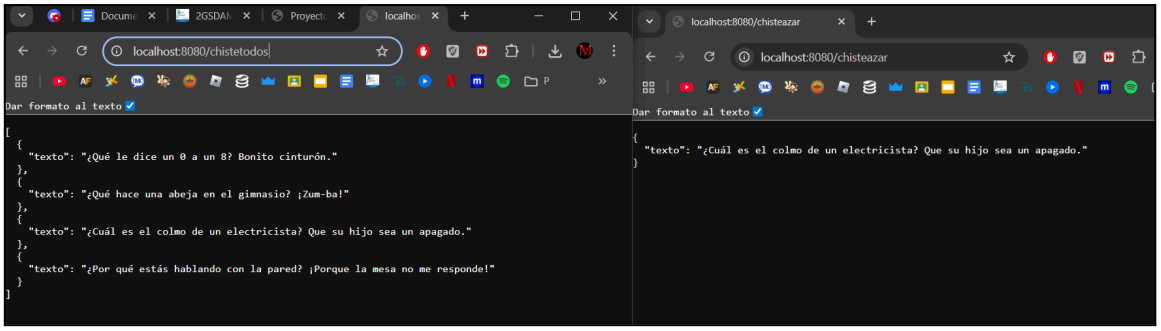
Podemos copiar directamente las carpetas que usamos para reutilizarlas y en el modelo no cambiaremos nada.

En la clase de servicio añadiremos un método que pase la lista de chiste al completo

```java
package com.example.demo.service;

import com.example.demo.modelo.*;

@Service
public class service {

    private List<Chiste> chistes;

    public service() {
        chistes = new ArrayList<>();
        chistes.add(new Chiste("¿Qué le dice un 0 a un 8? Bonito cinturón."));
        chistes.add(new Chiste("¿Qué hace una abeja en el gimnasio? ¡Zum-ba!"));
        chistes.add(new Chiste("¿Cuál es el colmo de un electricista? Que su hijo sea un apagado."));
        chistes.add(new Chiste("¿Por qué estás hablando con la pared? ¡Porque la mesa no me responde!"));
    }

    public Chiste obtenerChisteAleatorio() {
        return chistes.get((int) (Math.random() * chistes.size()));
    }
    public List<Chiste> todos() {
        return chistes;
    }
}
```

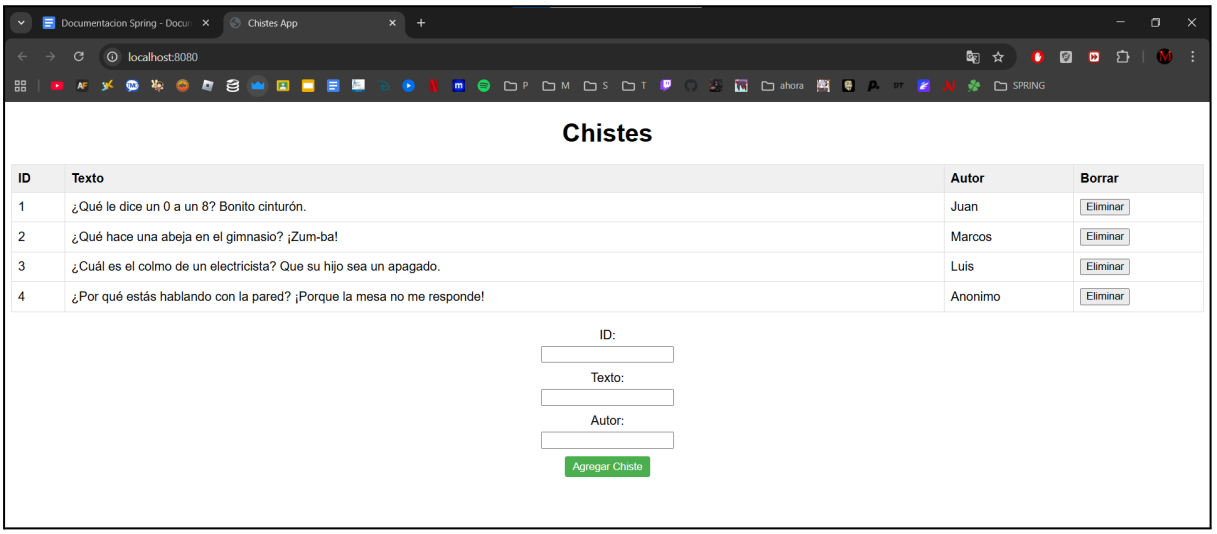
A diferencia del controlador vamos a cambiar 1 cosa y añadir 2 nuevos getmapping, deberemos cambiar el Controller por RestController.

```java

import com.example.demo.service.service;
import com.example.demo.modelo.Chiste;

import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class controlador {

    private service chisteService;

    public controlador(service chisteService) {
        this.chisteService = chisteService;
    }

    @GetMapping("/chisteazar")
    public Chiste chisteAzar() {
        Chiste chisteAzar = chisteService.obtenerChisteAleatorio();
        return chisteAzar;
    }

    @GetMapping("/chistetodos")
    public List <Chiste> chisteTodos() {
        List<Chiste> chistes=chisteService.todos();
        return chistes;
    }

}
```



API Funcional

Asi se veria una implementación de con html,js y css



Tareas propuesta

1-Replicar el ejemplo final dado.