



# spring<sup>®</sup>

## **SPRING**

INTRODUCCIÓN A SPRING

1

# INTRODUCCIÓN A SPRING

- Framework para el desarrollo de aplicaciones empresariales en Java.
- Incluye compatibilidad con Kotlin y Groovy.
- Requiere Java 17 a partir de Spring Framework 6.0.
- Es un framework de código abierto.
- Está dividido en módulos.
- Aparece en 2003.
- Además de Spring Framework, existen otros proyectos, como Spring Boot, Spring Security, Spring Data, Spring Cloud y Spring Batch, entre otros.
- Spring Framework 6.0 es compatible con :
  - Tomcat 10.1, Jetty 11 y Undertow 2.3 como servidores web
  - Hibernate ORM 6.1.

# INTRODUCCIÓN A SPRING



## SPRING BOOT

# INTRODUCCIÓN A SPRING

- Spring Boot es un framework de Java que simplifica la creación de aplicaciones basadas en el ecosistema Spring. Su principal objetivo es facilitar la configuración y el despliegue rápido de aplicaciones Spring, eliminando la necesidad de escribir mucha configuración manual.
- Antes de Spring Boot, configurar una aplicación con Spring podía ser complicado, ya que requería:
  - Muchos archivos XML para la configuración.
  - Configurar manualmente servidores como Tomcat.
  - Gestionar muchas dependencias a mano.

# INTRODUCCIÓN A SPRING

Spring Web y Spring MVC

# INTRODUCCIÓN A SPRING

- Spring Web es un módulo del framework Spring que proporciona soporte para el desarrollo de aplicaciones web en Java.
- Es uno de los componentes clave de Spring y se utiliza para crear aplicaciones basadas en arquitectura MVC (Modelo-Vista-Controlador).
- Características más relevantes:
  - Manejar peticiones HTTP.
  - Definir controladores (controllers).
  - Trabajar con formularios.
  - Enviar y recibir datos en formato JSON o XML.
  - Integrarse fácilmente con tecnologías como Thymeleaf, JSP, o APIs REST.

# INTRODUCCIÓN A SPRING

- Spring MVC (Modelo-Vista-Controlador) es la parte específica de Spring Web que implementa el patrón MVC:
  - Modelo (Model): Representa los datos y la lógica de negocio de la aplicación. Se gestiona mediante clases Java, servicios y repositorios (por ejemplo, con JPA o Hibernate).
  - Vista (View): Es la interfaz de usuario. Se suele crear con tecnologías como JSP, Thymeleaf u otros motores de plantillas. Se encarga de mostrar los datos del modelo.
  - Controlador (Controller): Gestiona las solicitudes HTTP. Interactúa con el modelo y devuelve una vista. Se anota con `@Controller` o `@RestController`.

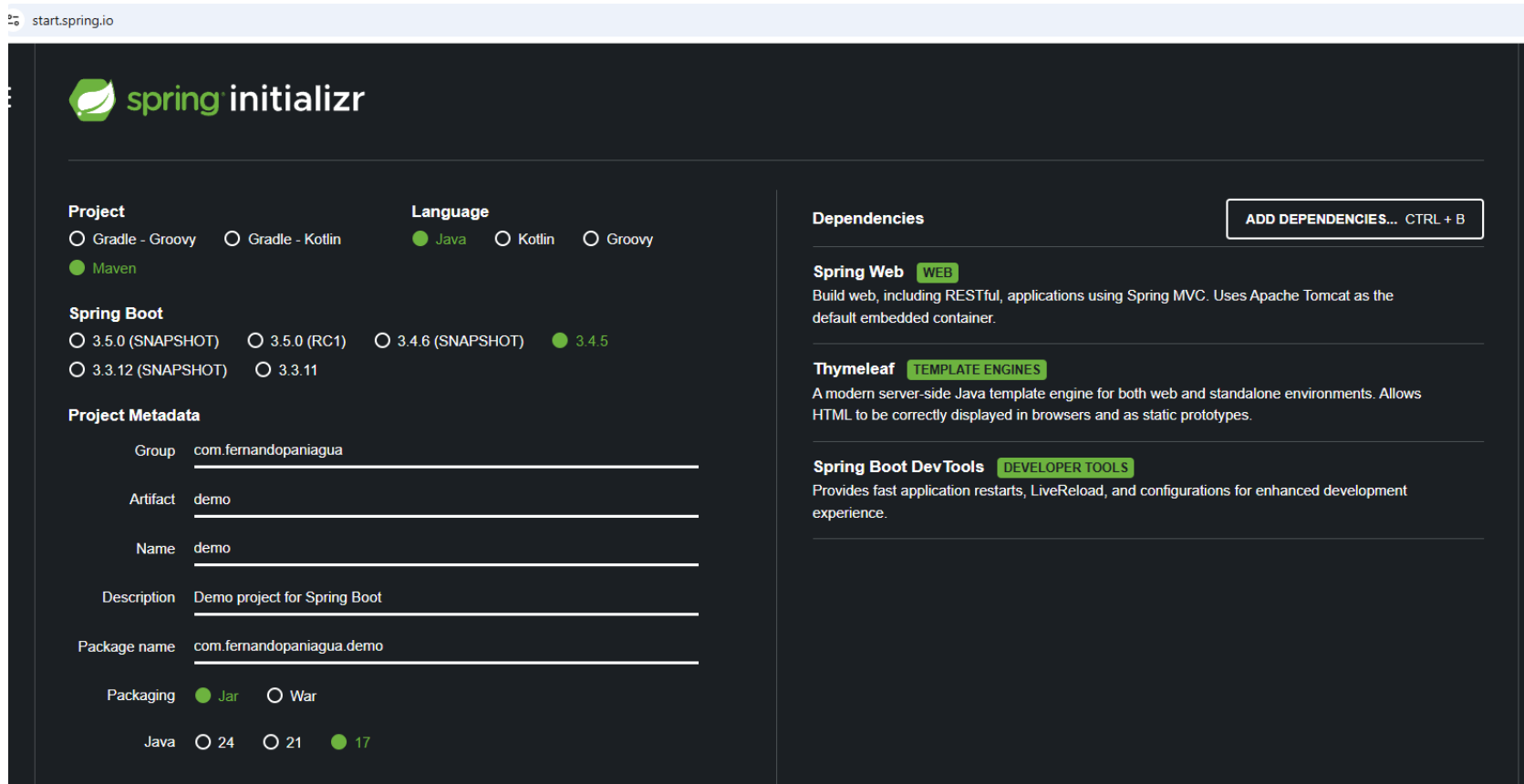
# INTRODUCCIÓN A SPRING

HOLA MUNDO



# INTRODUCCIÓN A SPRING

- Hello World utilizando Thymeleaf.
  - Generar el proyecto desde **spring** initializr.



The screenshot shows the Spring Initializr web application interface. The browser address bar displays "start.spring.io". The page features the "spring initializr" logo at the top left. Below the logo, there are sections for "Project", "Language", "Spring Boot", "Project Metadata", and "Dependencies".

**Project**

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ **Java** ☐ Kotlin ☐ Groovy

☒ **Maven**

**Spring Boot**

☐ 3.5.0 (SNAPSHOT) ☐ 3.5.0 (RC1) ☐ 3.4.6 (SNAPSHOT) ☒ **3.4.5**

☐ 3.3.12 (SNAPSHOT) ☐ 3.3.11

**Project Metadata**

Group

Artifact

Name

Description

Package name

Packaging ☒ **Jar** ☐ War

Java ☐ 24 ☐ 21 ☒ **17**

**Dependencies** ADD DEPENDENCIES... CTRL + B

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

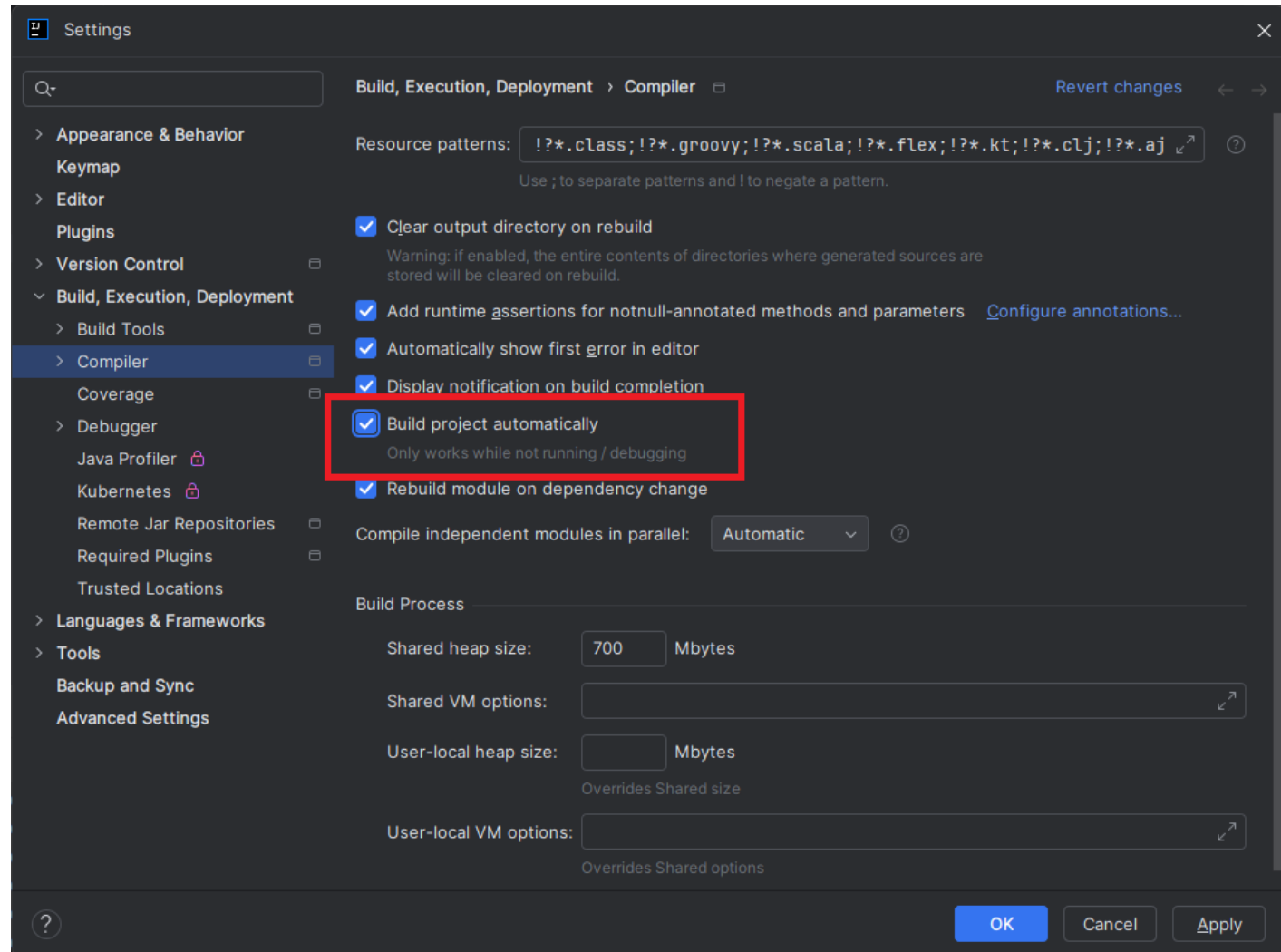
**Thymeleaf** TEMPLATE ENGINES  
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

**Spring Boot DevTools** DEVELOPER TOOLS  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

# INTRODUCCIÓN A SPRING



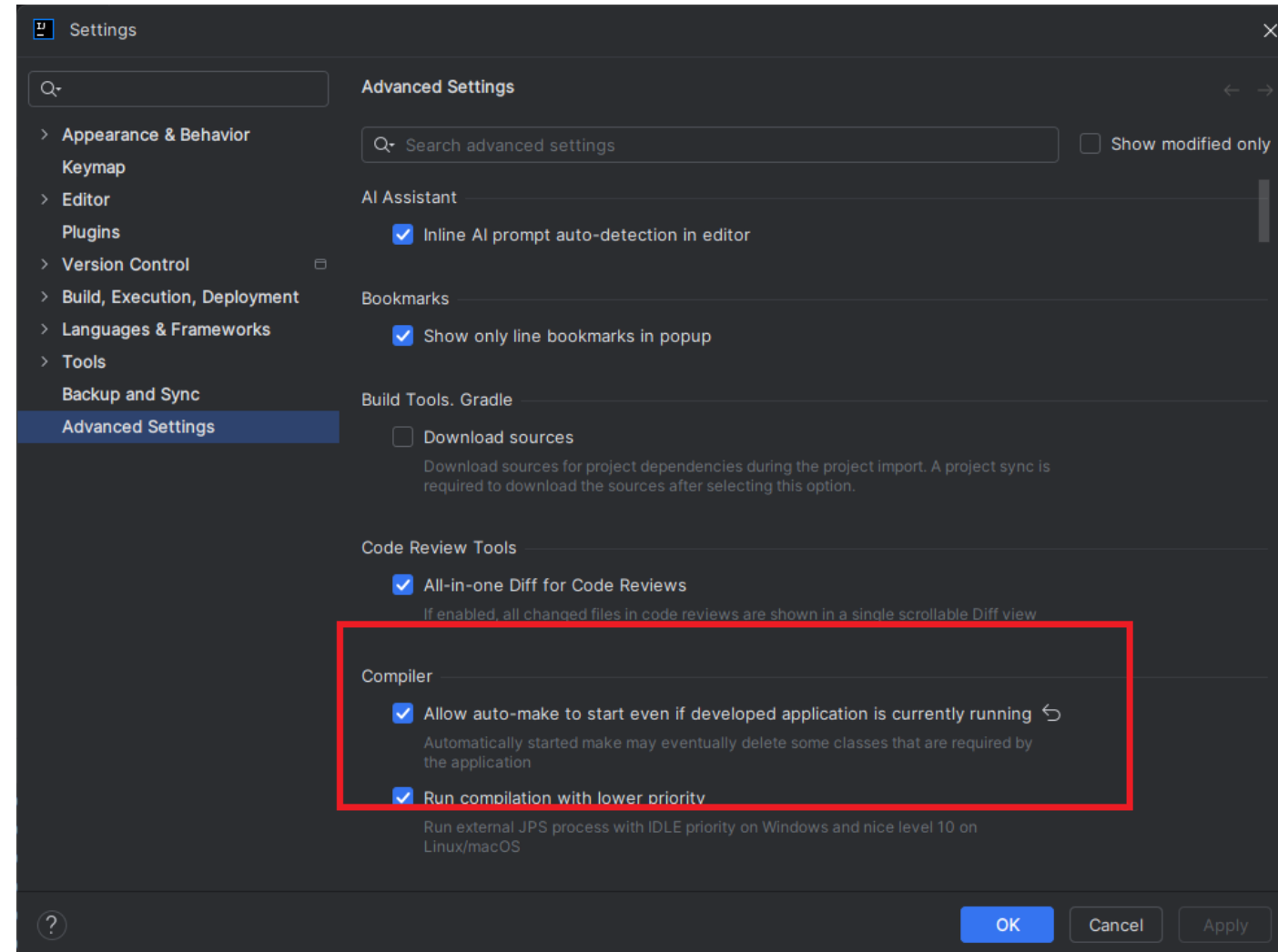
- Configurar el IDE.



# INTRODUCCIÓN A SPRING



- Configurar el IDE.



# INTRODUCCIÓN A SPRING

- Generar el fichero .zip.
- Descomprimir y abrir desde IDE.
- Verificar el *launcher* de la aplicación (creado automáticamente)

```
1 package com.fernandopaniagua.demo;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class DemoApplication {
8
9     public static void main(String[] args) {
10
11         SpringApplication.run(DemoApplication.class, args);
12     }
13
14 }
```

# INTRODUCCIÓN A SPRING

- Crear el controlador:

```
1 package com.fernandopaniagua.demo.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.GetMapping;
6
7 @Controller
8 public class HelloController {
9
10     @GetMapping("/")
11     public String hello(Model model) {
12         model.addAttribute("message", "¡Hola Mundo desde Spring Boot con Thymeleaf!");
13         return "hello";
14     }
15 }
```

# INTRODUCCIÓN A SPRING

- Crear la vista (en la carpeta resources/templates):

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <title>Hello</title>
5     <meta charset="UTF-8"/>
6 </head>
7 <body>
8 <h1 th:text="${message}">Mensaje por defecto</h1>
9 </body>
10 </html>
```

# INTRODUCCIÓN A SPRING

- Ejecutar el servidor y la aplicación:
  - `mvn spring-boot:run`
- Acceder a la aplicación (desde el navegador):
  - `http://localhost:8080/`

# INTRODUCCIÓN A SPRING



## Beans



# INTRODUCCIÓN A SPRING

- Un **bean** es un objeto que está gestionado por el contenedor de Spring. Es uno de los componentes fundamentales del framework.
- Forma parte del contexto de la aplicación y que es instanciado, ensamblado y gestionado completamente por el contenedor de Spring.
- Características:
  - Gestión del ciclo de vida: Spring controla la creación, inicialización y destrucción del bean.
  - Inyección de dependencias: Spring puede inyectar otros beans en un bean automáticamente.
  - Configuración flexible: Los beans pueden definirse de múltiples formas:
    - Anotaciones (@Component, @Service, @Repository, @Controller, @Configuration)
    - XML (en versiones más antiguas)
    - Clases de configuración Java (@Configuration + @Bean)

# INTRODUCCIÓN A SPRING

- Ciclo de vida de los beans. Beans *callbacks*.
  - Métodos especiales que se ejecutan automáticamente en ciertos momentos del ciclo de vida de un Spring Bean
  - `@PostConstruct`. Justo después de ser creado e inyectado (post-inicialización).
  - `@PreDestroy`. Justo antes de ser destruido (pre-destrucción).

```
@PostConstruct
public void postConstruct(){
    System.err.println("Ejecutando postConstruct de MovieService");
}
```

```
@PreDestroy
public void preDestroy(){
    System.err.println("Ejecutando postConstruct de MovieService");
}
```

# INTRODUCCIÓN A SPRING

- Tipos de beans:
  - **@Component**. Bean genérico. @Service, @Repository, @Controller y @Configuration son especializaciones. Se debe utilizar @Component cuando ninguno de las otras anotaciones encaja.
  - **@Service**. Un servicio. Indica que la clase pertenece a la capa de servicios (lógica de aplicación).
  - **@Repository**. Determina que una clase tiene acceso a la base de datos. Activa el manejo automático de excepciones, convirtiendo las generadas por la persistencia en DataAccessException de Spring. Implementa el DAO.
  - **@Controller**. Controlador de Spring MVC. Intercepta las llamadas HTTP.
  - **@Configuration**. Contiene definiciones de beans.
- Todos son detectados automáticamente por Spring ya que incluyen la anotación @ComponentScan.

# INTRODUCCIÓN A SPRING

- Inyección de componentes.
- 3 formas:
  - Por constructor (recomendada).
  - Por setter.
  - Inyección directa.
- Ejemplo:
  - Componente:

```
@Component
public class Saludador {
    public void saludar(){
        System.out.println("Hola");
    }
}
```

# INTRODUCCIÓN A SPRING

- Inyección de componentes.
  - Inyección por constructor:

```
@Controller
public class HelloController {
    private final Saludador saludador;
    public HelloController(Saludador saludador) {
        this.saludador = saludador;
    }
}
```

```
@GetMapping("/hello-world")
public String hello(Model model) {
    this.saludador.saludar();
    model.addAttribute("message", "¡Hola Mundo desde Spring Boot con Thymeleaf!");
    return "hello";
}
}
```

# INTRODUCCIÓN A SPRING

- Inyección de componentes.
  - Inyección por setter:

```
private Saludador saludador;
```

```
@Autowired
```

```
public void setSaludador(Saludador saludador) {  
    this.saludador = saludador;  
}
```

```
@GetMapping("/hello-world")
```

```
public String hello(Model model) {
```

```
    this.saludador.saludar();
```

```
    model.addAttribute("message", "¡Hola Mundo desde Spring Boot con Thymeleaf!");  
    return "hello";
```

```
}
```

# INTRODUCCIÓN A SPRING

- Inyección de componentes.
  - Inyección directa:

```
@Autowired
```

```
private Saludador saludador;
```

```
@GetMapping("/hello-world")  
public String hello(Model model) {
```

```
    this.saludador.saludar();
```

```
    model.addAttribute("message", "¡Hola Mundo desde Spring Boot con Thymeleaf!");  
    return "hello";
```

```
}
```

# INTRODUCCIÓN A SPRING

- Inyección de componentes contruidos en @Configuration.

```
public class TestService {  
    public void writeMessage(String message) {  
        System.err.printf("El mensaje %s ha sido escrito  
desde ServiceTest",message);  
    }  
}
```

Clase con funcionalidad (no necesariamente es un componente)

```
@Configuration  
public class ConfigurationTest {  
    @Bean  
    public TestService testService(){  
        return new TestService();  
    }  
}
```

La anotación @Configuration en Spring se usa para marcar una clase como una clase de configuración, es decir, una clase que define uno o más beans que deben ser gestionados por el contenedor de Spring.

Se puede inyectar la dependencia al servicio

```
@Autowired  
TestService testService;
```



# INTRODUCCIÓN A SPRING



Anotaciones

# INTRODUCCIÓN A SPRING

Anotación	Ámbito	Descripción
@SpringBootApplication	Clase	Marca la clase de arranque de la aplicación spring.
@Component	Clase	Marca la clase como un componente.
@Controller	Clase	Marca la clase como un componente de tipo controlador.
@Service	Clase	Marca la clase como un servicio.
@Repository	Clase	Marca la clase como repositorio.
@Configuration	Clase	Marca la clase como un componente de configuración.
@Bean	Método	Marca el método como vean gestionado por spring.
@PostConstruct	Método	Indica que el método se debe ejecutar después de construir el componente.
@PreDestroy	Método	Indica que el método se debe ejecutar antes de destruir el componente.
@Autowired	Atributo	Inyecta una instancia del tipo de la clase del atributo.
@RequestMapping	Clase/método	Determina el path general de las peticiones gestionadas por la clase.
@GetMapping	Método	Determina el path particular de las peticiones gestionadas por el método (tipo GET).
@PostMapping	Método	Determina el path particular de las peticiones gestionadas por el método (tipo POST).
@PutMapping	Método	Determina el path particular de las peticiones gestionadas por el método (tipo PUT).
@DeleteMapping	Método	Determina el path particular de las peticiones gestionadas por el método (tipo DELETE).
@RequestParam	Parámetro	Permite obtener un parámetro de la llamada HTTP
@ModelAttribute	Parámetro	Permite obtener una entidad como parámetro de la llamada HTTP
@PathVariable	Parámetro	Permite obtener un parámetro incluido en la URL de la llamada HTTP

# INTRODUCCIÓN A SPRING

Anotación	Ámbito	Descripción
@Aspect	Clase	Marca una clase como aspecto para la POA.
@After, @Before, @Around	Método	Determina en qué momento se va a ejecutar un método de un aspecto.
@EventListener	Método	Determina que el método realiza la escucha activa de eventos.
@Entity	Clase	Marca la clase como entidad (persistencia).
@Id	Atributo	Indica que el atributo es clave de la entidad.
@GeneratedValue	Atributo	Indica la estrategia de generación del valor del atributo.
@Query	Método	Asocia una consulta JPQL, SQL o de tipo proyección al método.
@Column	Atributo	Determina que el atributo se corresponde con una columna de la tabla.
@OneToOne, @OneToMany, @ManyToOne, @ManyToMany	Atributo	Establece la relación con otras entidades.
@JoinColumn	Atributo	Determina una clave foránea.
@Transient	Atributo	Indica que el atributo no es persistente.
@Embeddable	Clase	Indica que la clase puede ser embebida en otra entidad.
@Embedded	Atributo	Indica que el atributo es embebido en otra entidad.

# INTRODUCCIÓN A SPRING

## Spring MVC

# INTRODUCCIÓN A SPRING

## •Spring MVC

### •Componentes clave:

- Controlador. `@Controller`. Intercepta las peticiones.
- Servicio. `@Service`. Implementa la lógica de negocio.
- Repositorio. `@Repository`. Contiene la persistencia.
- Vista. HTML + anotaciones. Thymeleaf como opción recomendada.

### •Las peticiones tienen el siguiente flujo:

- Cliente → Controlador → Servicio → Repositorio → Servicio → Controlador → Vista.

# INTRODUCCIÓN A SPRING

•Controlador:

**@Controller**

**@RequestMapping("/movie")**

```
public class MovieController {  
    private final MovieService movieService;  
    public MovieController(MovieService movieService) {  
        this.movieService = movieService;  
    }  
}
```

**@PostMapping("/create")**

```
public String createMovie(@ModelAttribute Movie movie){  
    Movie newMovie = this.movieService.create(movie);  
    return "redirect:/";  
}
```

# INTRODUCCIÓN A SPRING



## .Controlador:

### .Los datos se pueden recuperar:

```
@PostMapping("/create")
public String createMovie(@ModelAttribute Movie movie){
    Movie newMovie = this.movieService.create(movie);
    return "redirect:/";
}
```

```
@GetMapping("/delete/{id}")
public String deleteMovie(@PathVariable int id){
    this.movieService.delete(this.movieService.read(id));
    return "redirect:/";
}
```

```
@GetMapping("/search")
public String search(Model model, @RequestParam(required = false) String director){
    List<String> directors = this.movieService.getDirectorNames();
    List<Movie> movies = null;
```

# INTRODUCCIÓN A SPRING



•Servicio:

•Contiene la lógica de negocio y utiliza los repositorios:

**@Service**

```
public class MovieService {
```

```
    @Autowired
```

```
    IMovieRepository movieRepository;
```

```
    public Movie create(Movie movie){
```

```
        Movie newMovie = movieRepository.save((movie));
```

```
        return newMovie;
```

```
    }
```



# INTRODUCCIÓN A SPRING

## •Repositorio:

- Es una interface que incluye código implícito.

- Contiene el DAO:

```
public interface IMovieRepository extends JpaRepository<Movie, Integer> {  
    //Solución basada en anotación + JPQL  
    @Query("SELECT DISTINCT m.director FROM Movie m")  
    List<String> getDirectorNames();  
  
    //Generado por Spring Data JPA automáticamente en tiempo de ejecución  
    List<Movie> findByDirector(String name);  
  
    //Generado por Spring Data JPA automáticamente en tiempo de ejecución  
    void deleteByDirector(String name);  
}
```

# INTRODUCCIÓN A SPRING



## •Repositorio:

### •Algunos ejemplos de métodos implícitos:

- `findByEdad(int edad)` → `SELECT u FROM Usuario u WHERE u.edad = :edad`
- `findByNombreAndEdad(String, int)` → `WHERE nombre = ? AND edad = ?`
- `findByNombreContaining(String)` → `WHERE nombre LIKE %?%`
- `findTop3ByEdadGreaterThan(int)` → `WHERE edad > ? ORDER BY edad DESC LIMIT 3`

# INTRODUCCIÓN A SPRING



Eventos

# INTRODUCCIÓN A SPRING

- Mecanismo de comunicación desacoplado entre componentes de la aplicación spring.
- Consta de:
  - Evento. Clase realizable que se propaga por la aplicación (POJO en versiones modernas de spring, heredada de `ApplicationEvent` en versiones antiguas).
  - Publisher. Emisor del evento.
  - Listener(s). Receptor(es) del evento.

# INTRODUCCIÓN A SPRING

• El evento es una clase “convencional”:

```
public class CreationEvent {  
    private String message;  
  
    public CreationEvent(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

# INTRODUCCIÓN A SPRING

• El “Publisher” es un componente con referencias a un objeto `ApplicationEventPublisher`:

```
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Component;

@Component
public class EventPublisher {
    private final ApplicationEventPublisher publisher;

    public EventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void publishEvent(String message) {
        CreationEvent creationEvent = new CreationEvent(message);
        publisher.publishEvent(creationEvent);
    }
}
```

# INTRODUCCIÓN A SPRING

## • Generador del evento:

- Se inyecta la referencia al Publisher:

```
@Autowired  
EventPublisher eventPublisher;
```

- Se lanza el evento a través del publicador:

```
eventPublisher.publishEvent(String.format("Película %s creada", movie.getTitle()));
```

## • Consumidor del evento:

```
@EventListener  
public void manejarEvento(CreationEvent event) {  
    System.out.println("Evento recibido: " + event.getMessage());  
}
```

# INTRODUCCIÓN A SPRING

Programación Orientada a Aspectos (POA)  
Aspect-oriented programming (AOP)



# INTRODUCCIÓN A SPRING

- Permite encapsular “preocupaciones” transversal (*cross-cutting concerns*). Es decir, encapsular código que resuelve algún problema/necesidad que afecta a lo largo del proyecto.
- Un aspecto es la encapsulación de un bloque de código.
- Se puede ejecutar antes o después del método al que se asocia.
- AOP solo intercepta métodos públicos no estáticos por defecto
- Conceptos:
  - **Aspecto** (Aspect): Módulo que encapsula una preocupación transversal.
  - **Join Point**: Puntos específicos del flujo de ejecución del programa (por ejemplo, una llamada a un método).
  - **Advice**: Código que se ejecuta en un join point. Puede ser:
    - **Before**: antes de ejecutar el método.
    - **After**: después de ejecutarlo.
    - **Around**: envolviendo la ejecución del método. El más flexible.
  - **Pointcut**: Expresión que define en qué join points se aplicará el advice.
  - **Weaving**: Es el proceso de aplicar los aspectos a los join points. En Spring AOP, el weaving se realiza en tiempo de ejecución (runtime), lo que significa que los aspectos se aplican mientras la aplicación se está ejecutando..

# INTRODUCCIÓN A SPRING

Librería:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96


97

98

99

100

Last Release on Apr 17, 2025




2. **Spring Boot Starter AOP**

[org.springframework.boot » spring-boot-starter-aop](https://org.springframework.boot/spring-boot-starter-aop)

Starter for aspect-oriented programming with Spring AOP and AspectJ

Last Release on Apr 24, 2025



3. **JBoss AOP Framework**

# INTRODUCCIÓN A SPRING

## •Aspecto:

- `joinPoint` → Tiene información sobre el punto en el que se ha “insertado” el aspecto.
- `advice` → El código que se ejecuta.
- `pointcut` → En qué punto concreto se ejecuta el código.

`@Aspect`

`@Component`

`public class LogAspect {`

`@After("execution(* com.fernandopaniagua democrudspring.controller.*(..))")`

`public void writeLog(JoinPoint joinPoint) {`

`LocalDateTime timestamp = LocalDateTime.now();`

`DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");`

`String formattedTimestamp = timestamp.format(formatter);`

`System.err.printf("Ejecutando %s. %s %n", joinPoint.getSignature().getName(), formattedTimestamp);`

`}`

`}`

# INTRODUCCIÓN A SPRING



Thymeleaf

# INTRODUCCIÓN A SPRING

- Thymeleaf es un motor de plantillas para Java, muy utilizado en aplicaciones web basadas en Spring. Su función principal es generar páginas HTML dinámicas desde el backend Java, reemplazando datos y estructuras directamente dentro del HTML.
- Thymeleaf es un moderno motor de plantillas Java del lado del servidor para entornos web e independientes.
- Los ficheros tienen **extensión html** y se ubican en la carpeta **resources/templates**.
- <https://www.thymeleaf.org/>
- <https://docs.spring.io/spring-framework/reference/web/webmvc-view/mvc-thymeleaf.html>
- Otras tecnologías de vistas:
  - <https://docs.spring.io/spring-framework/reference/web/webmvc-view.html>

# INTRODUCCIÓN A SPRING

- Etiquetas principales:

Atributo	Descripción
th:text	Sustituye el contenido del elemento con texto evaluado.
th:utext	Igual que th:text, pero permite HTML sin escapar.
th:if	Renderiza el elemento solo si la condición es verdadera.
th:unless	Renderiza el elemento solo si la condición es falsa.
th:each	Bucle para iterar sobre listas u objetos.
th:href	Establece el atributo href de un enlace con valores dinámicos.
th:src	Establece la fuente (src) de imágenes u otros elementos multimedia.
th:attr	Permite definir uno o varios atributos dinámicos en un solo lugar.
th:value	Asigna el valor de un campo de formulario (input, textarea, etc.).
th:object	Define el objeto base para los campos de un formulario.
th:field	Usado dentro de formularios para enlazar campos con atributos del objeto.

# INTRODUCCIÓN A SPRING

- Código de ejemplo:

```
<!DOCTYPE html>  
<html lang="es" xmlns:th="http://www.thymeleaf.org">  
<head>  
  
<select class="custom-select" name="director" id="director-select">  
  <option value="all" selected>Todos</option>  
  <option th:each="director : ${directors}" th:text="${director}" th:value="${director}"></option>  
</select>
```

# INTRODUCCIÓN A SPRING

- Código de ejemplo:

```
<tbody>
  <tr th:each="movie : ${movies}">
    <td th:text="${movie.id}"></td>
    <td th:text="${movie.title}"></td>
    <td th:text="${movie.director}"></td>
    <td>
      
    </td>
    <td th:text="${movie.year}"></td>
    <td th:text="${movie.rating}"></td>
    <td>
      <a th:href="@{/movie/delete/{id}(id=${movie.id})}">
        
      </a>
      <a th:href="@{/update/{id}(id=${movie.id})}">
        
      </a>
    </td>
  </tr>
</tbody>
```



# INTRODUCCIÓN A SPRING

- Código de ejemplo. Fragmentos.
- Definición del fragmento: fichero `fragment_navbar.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head></head>
<body>
<nav th:fragment="navBar">
  <ul class="navbar">
    <li><a href="/create">Create</a></li>
    <li><a href="/">Movies</a></li>
    <li><a href="/movie/search">Search</a></li>
  </ul>
</nav>
</body>
</html>
```

# INTRODUCCIÓN A SPRING

- Código de ejemplo. Fragmentos.
- Uso del fragmento

```
<span th:replace="fragment_navbar :: navbar"></span>
```

- Resultado:

El contenedor es el del fragmento, no el del código que lo inserta. En este caso es **nav** (no span)

```
.. <nav> == $0
  <ul class="navbar"> flex
    <li>
      <a href="/create">...</a>
    </li>
    <li>
      <a href="/">...</a>
    </li>
    <li>
      <a href="/movie/search">...</a>
    </li>
  </ul>
</nav>
```

# INTRODUCCIÓN A SPRING



Servicios web Hola Mundo

# INTRODUCCIÓN A SPRING

- CONSTRUCCIÓN DE UN SERVICIO WEB **RESTful**.

- Petición GET en <http://localhost:8080/saludo>

- Respuesta: {"id":1,"mensaje":"Hola, mundo!"}

- Petición GET en <http://localhost:8080/saludo?nombre=Melissa>

- Respuesta: {"id":1,"mensaje":"Hola, Melissa!"}

- ¿Qué necesitamos?

- Java 17 o posterior.

- IDE.

- Gradle 7.5+ o Maven 3.5+

# INTRODUCCIÓN A SPRING

- CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.
  - Creación de proyecto.
    - <https://start.spring.io>
    - Elegir **MAVEN** y **JAVA**.
    - Elegir **SPRING WEB** como dependencia.

# INTRODUCCIÓN A SPRING

• CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.

• Entidad:

```
public record Saludo(long id, String mensaje) { }
```

• Controller:

```
@RestController
public class SaludadorController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @GetMapping("/saludo")
    public Saludo saludo(@RequestParam(value = "nombre", defaultValue = "World") String name) {
        return new Saludo(counter.incrementAndGet(), String.format(template, name));
    }
}
```

# INTRODUCCIÓN A SPRING

## • CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.

• Alternativa 0. Ejecutando desde el IDE.

• Alternativa 1. Directamente desde **Maven**

• **mvn spring-boot:run**

• Alternativa 2. Compilar desde Maven y ejecutar el fichero .jar:

• **mvn clean package**

• Genera el fichero **jar** en la carpeta **target**.

• Ejecutar el proyecto:

• **java -jar target\demo-0.0.1-SNAPSHOT.jar**

• Probar el WS:

• Desde el navegador: <http://localhost:8080/saludo?nombre=Fernando>

• Desde un terminal: `curl https://api.ejemplo.com/recurso`

# INTRODUCCIÓN A SPRING



Despliegue



# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.

### .Crear fichero **Dockerfile** en la carpeta raíz del proyecto

```
FROM maven:3.9.0-eclipse-temurin-19 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests
```

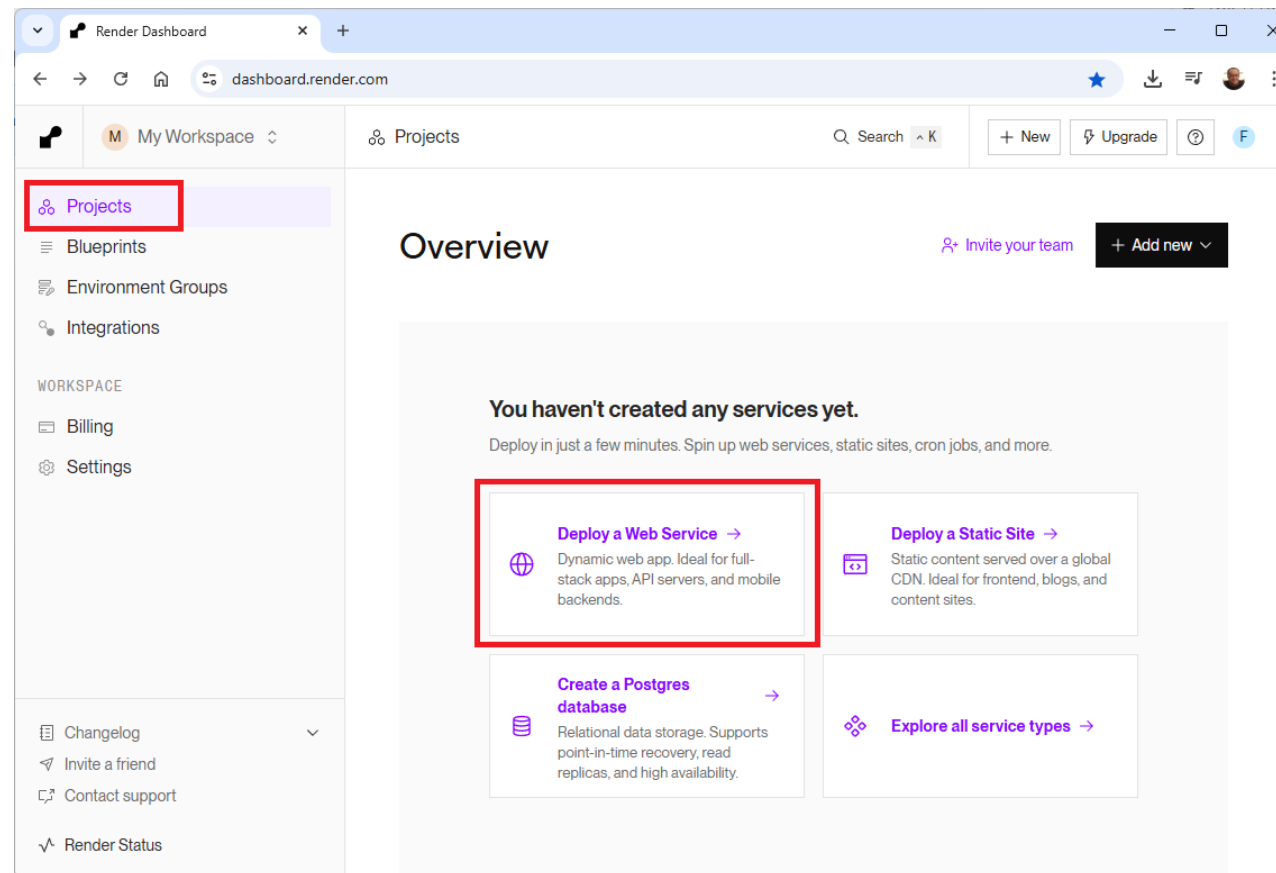
```
# Etapa final: solo el runtime
FROM openjdk:19-jdk
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","app.jar"]
```

### .Subir el repositorio a GitHub.

# INTRODUCCIÓN A SPRING

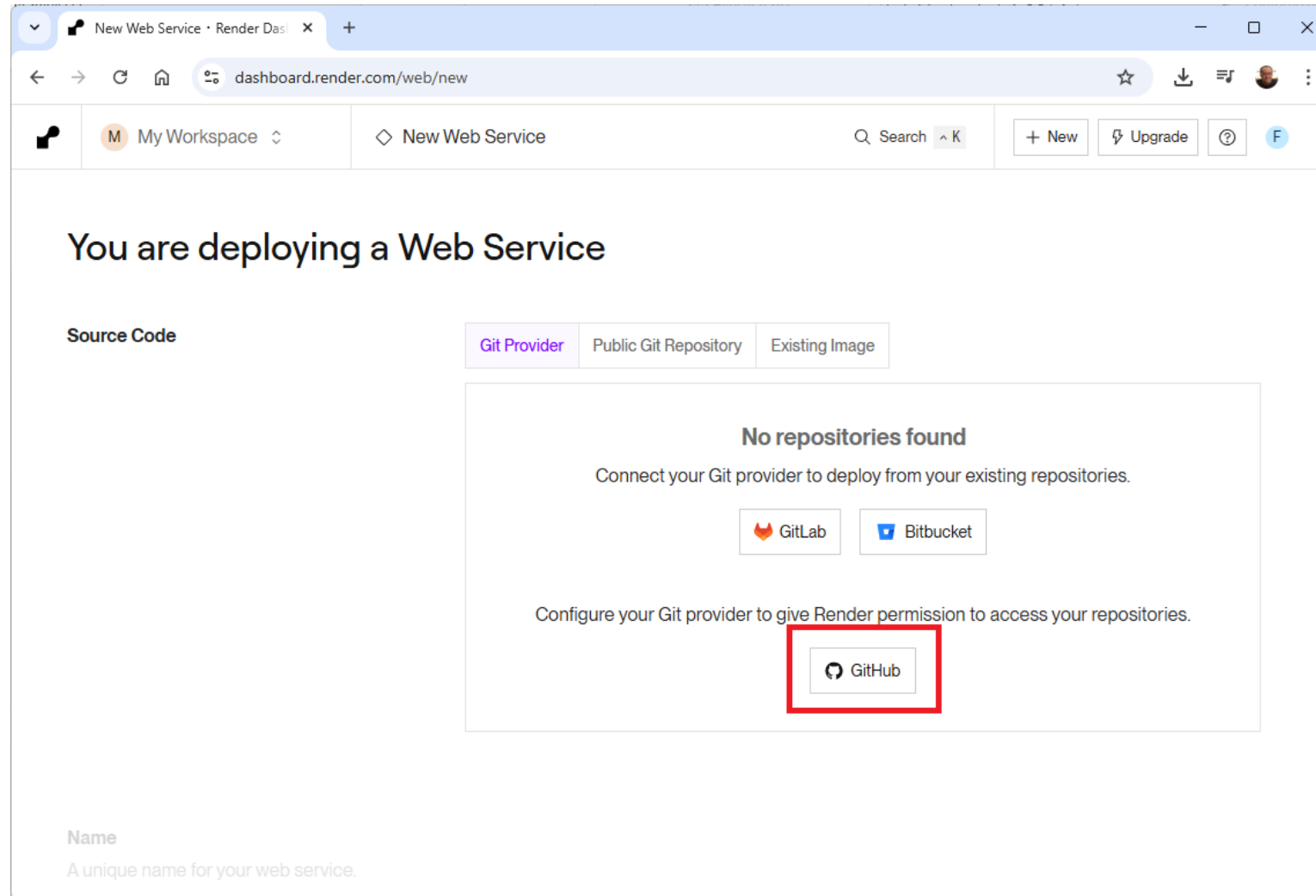
• CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.

• Alojamiento en <https://render.com/>



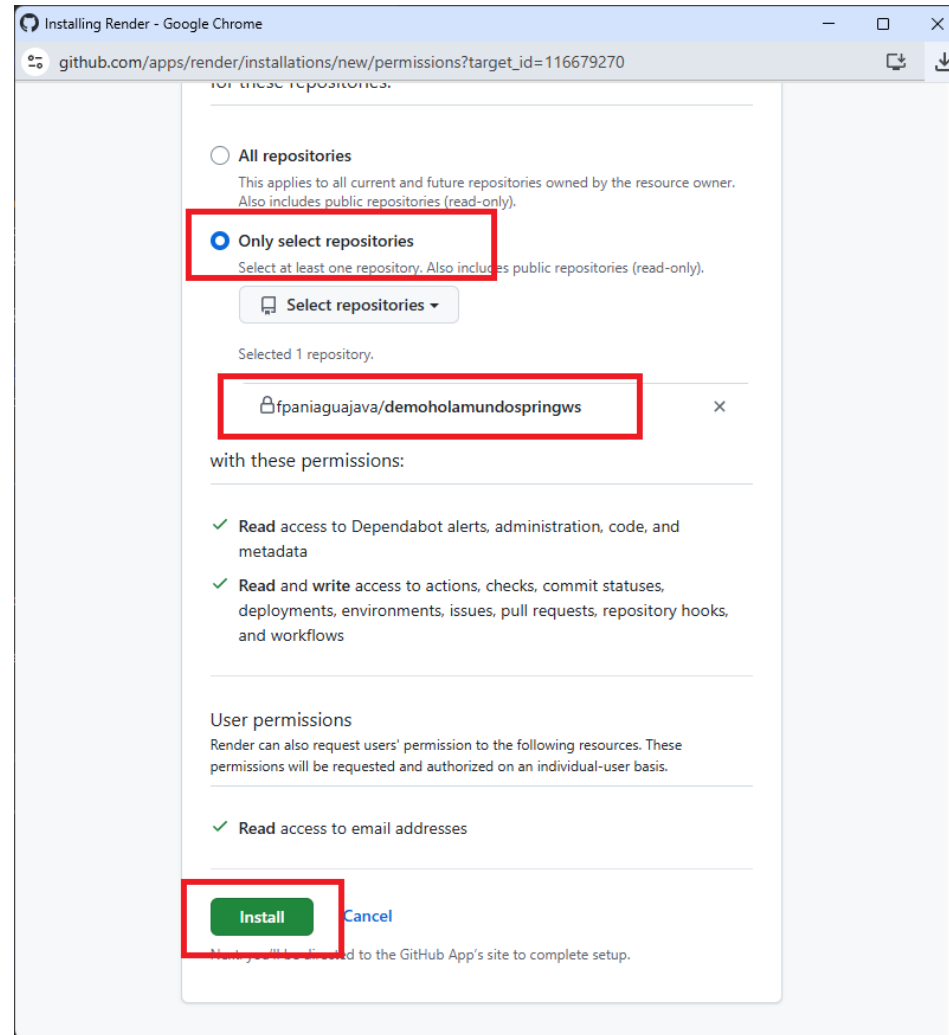
# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



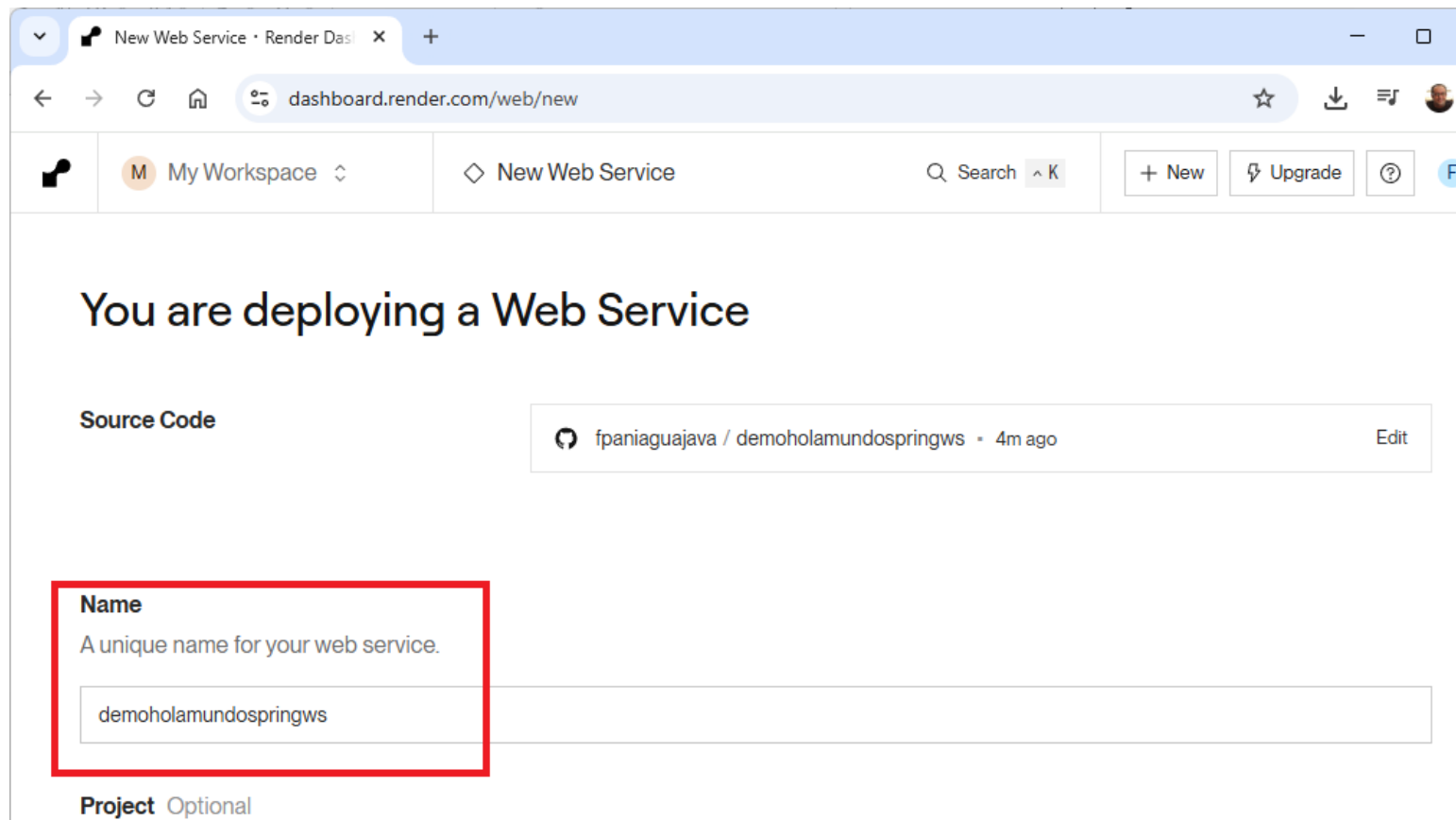
# INTRODUCCIÓN A SPRING

## • CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



Render Dashboard: New Web Service

Source Code: fpaniguajava / demoholamundospringws - 4m ago

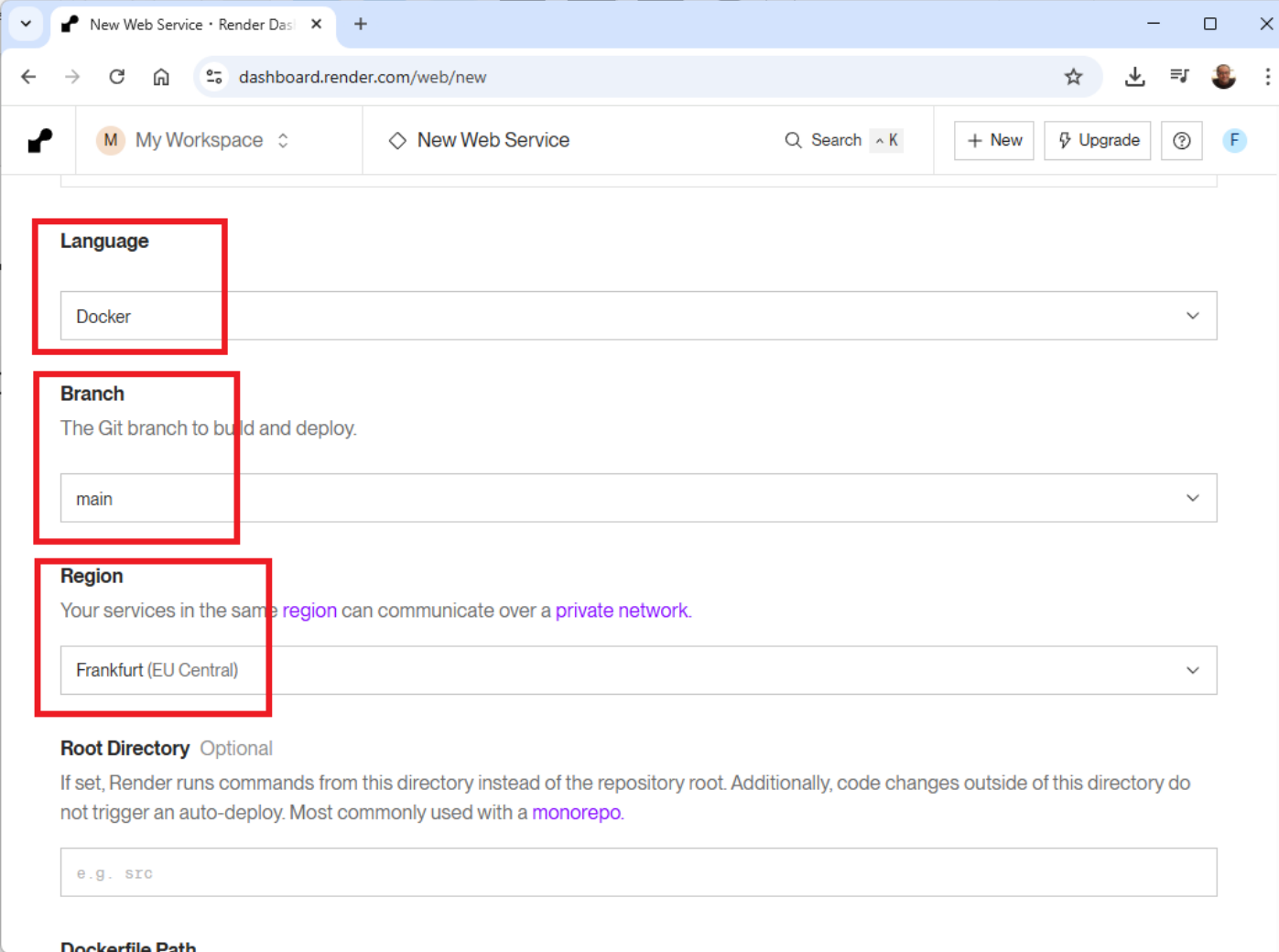
**Name**  
A unique name for your web service.

demoholamundospringws

**Project** Optional

# INTRODUCCIÓN A SPRING

## • CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



New Web Service • Render Dashboard

dashboard.render.com/web/new

My Workspace New Web Service Search ^ K + New Upgrade ? F

**Language**

Docker

**Branch**

The Git branch to build and deploy.

main

**Region**

Your services in the same region can communicate over a private network.

Frankfurt (EU Central)

**Root Directory** Optional

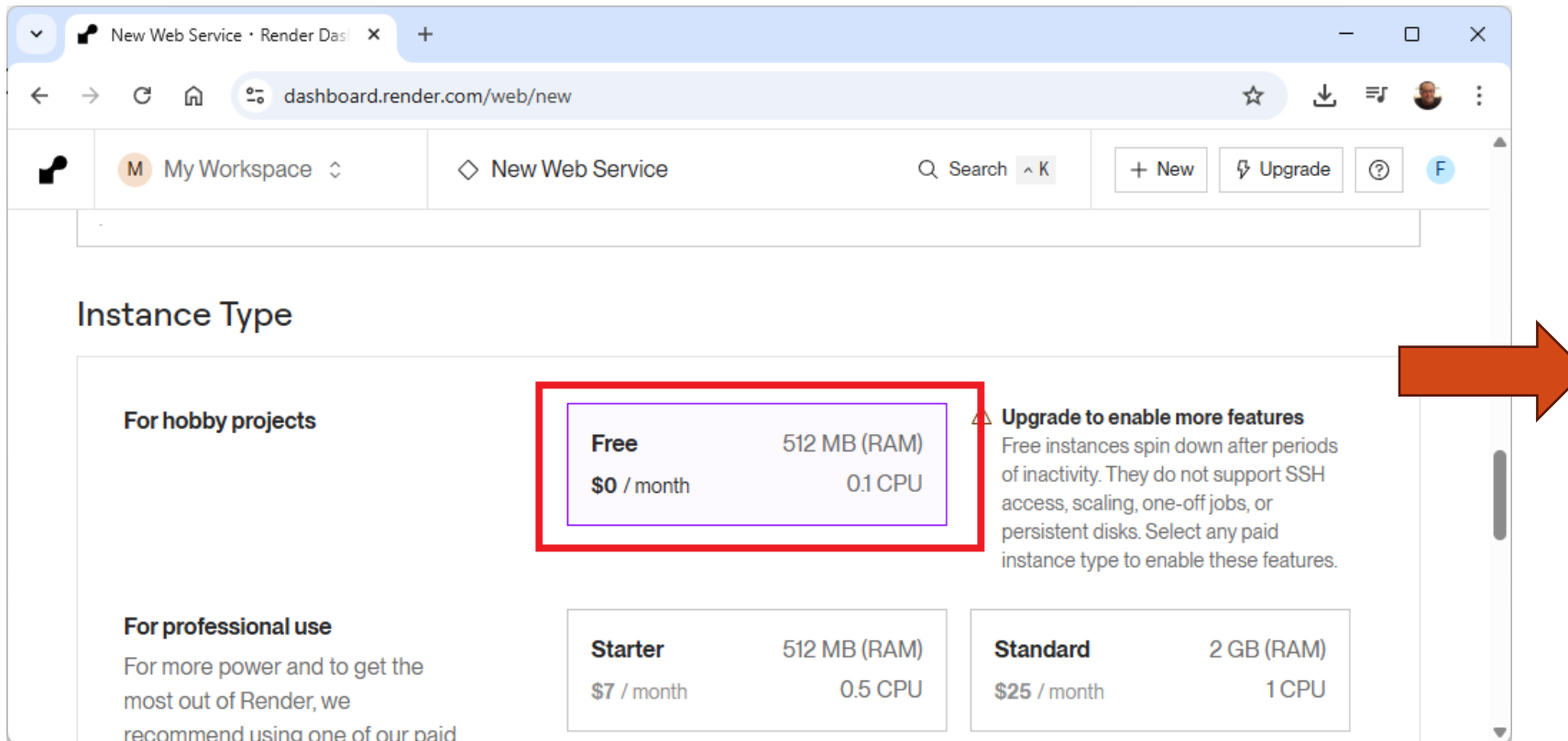
If set, Render runs commands from this directory instead of the repository root. Additionally, code changes outside of this directory do not trigger an auto-deploy. Most commonly used with a monorepo.

e.g. src

**Dockerfile Path**

# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



dashboards.render.com/web/new

My Workspace New Web Service

Search

+ New Upgrade ? F

### Instance Type

**For hobby projects**

Instance Type	Price	RAM	CPU
<b>Free</b>	<b>\$0 / month</b>	512 MB (RAM)	0.1 CPU

**Upgrade to enable more features**  
Free instances spin down after periods of inactivity. They do not support SSH access, scaling, one-off jobs, or persistent disks. Select any paid instance type to enable these features.

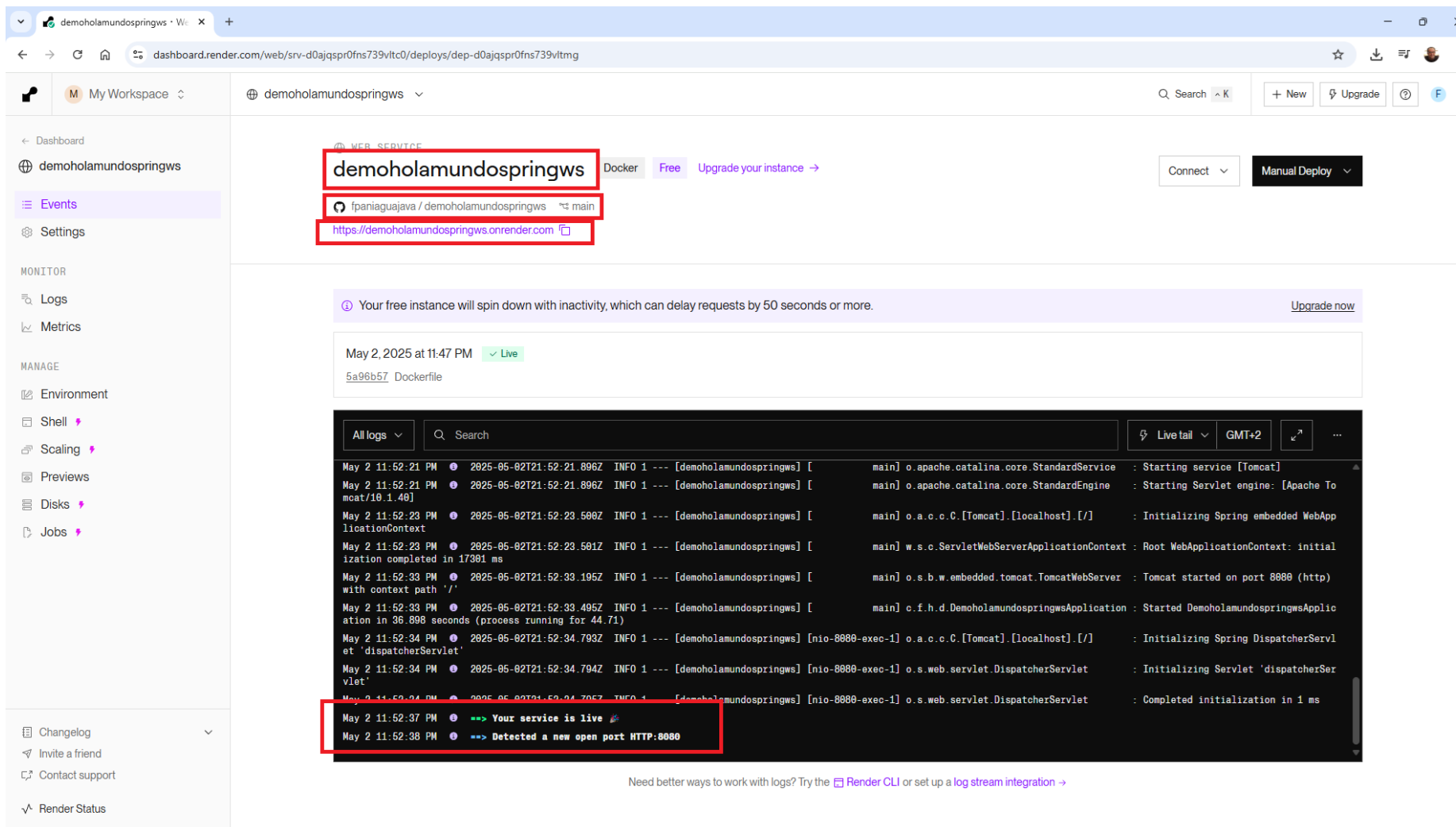
**For professional use**  
For more power and to get the most out of Render, we recommend using one of our paid

Instance Type	Price	RAM	CPU
<b>Starter</b>	<b>\$7 / month</b>	512 MB (RAM)	0.5 CPU
<b>Standard</b>	<b>\$25 / month</b>	2 GB (RAM)	1 CPU

Deploy Web Service

# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



The screenshot shows the Render dashboard for a service named **demoholamundospringws**. The service is running on Docker and is currently in a "Free" state. The dashboard includes a sidebar with navigation options like Dashboard, Events, Settings, and MONITOR. The main content area displays the service details, including the repository `fpaniguajava / demoholamundospringws` and the deployment URL `https://demoholamundospringws.onrender.com`. A notification indicates that the free instance will spin down with inactivity. Below this, the deployment logs are shown, detailing the startup process of the Spring application. The logs include the following entries:

```
May 2 11:52:21 PM [demoholamundospringws] [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
May 2 11:52:21 PM [demoholamundospringws] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.40]
May 2 11:52:23 PM [demoholamundospringws] [main] o.a.c.o.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
May 2 11:52:23 PM [demoholamundospringws] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 17301 ms
May 2 11:52:33 PM [demoholamundospringws] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
May 2 11:52:33 PM [demoholamundospringws] [main] c.f.h.d.DemoholamundospringwsApplication : Started DemoholamundospringwsApplication in 36.898 seconds (process running for 44.71)
May 2 11:52:34 PM [demoholamundospringws] [nio-8080-exec-1] o.a.c.o.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
May 2 11:52:34 PM [demoholamundospringws] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
May 2 11:52:34 PM [demoholamundospringws] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
May 2 11:52:37 PM [demoholamundospringws] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
May 2 11:52:38 PM [demoholamundospringws] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
```

The logs also show a message indicating that the service is live and a new open port HTTP:8080 has been detected.



# INTRODUCCIÓN A SPRING

## .CONSTRUCCIÓN DE UN SERVICIO WEB RESTful.



# INTRODUCCIÓN A SPRING



WS CRUD

# INTRODUCCIÓN A SPRING

•Librerías:

## Dependencies

ADD DEPENDENCIES... CTRL + B

### Spring Web **WEB**

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

### Spring Data JPA **SQL**

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

### PostgreSQL Driver **SQL**

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

### Spring Boot DevTools **DEVELOPER TOOLS**

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

### Rest Repositories **WEB**

Exposing Spring Data repositories over REST via Spring Data REST.

# INTRODUCCIÓN A SPRING

## •application.properties

```
spring.application.name=springwscrud
```

```
# Puerto del servidor (opcional)
```

```
server.port=8080
```

```
# Configuración de la base de datos PostgreSQL
```

```
spring.datasource.url=jdbc:postgresql://localhost:5432/aves
```

```
spring.datasource.username=user_aves
```

```
spring.datasource.password=password_aves
```

```
spring.datasource.driver-class-name=org.postgresql.Driver
```

```
# Configuración JPA / Hibernate
```

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

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

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

```
# Log del Hikari (Pool de conexiones)
```

```
# Con este atributo en modo DEBUG da información del POOL
```

```
logging.level.com.zaxxer.hikari=DEBUG
```

```
# Pool
```

```
spring.datasource.hikari.maximum-pool-size=10
```

```
spring.datasource.hikari.minimum-idle=5
```

# INTRODUCCIÓN A SPRING

## .Modelos

```
@Entity
@Table(name="familias")
public class Familia {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "nombre")
    private String nombre;
```

# INTRODUCCIÓN A SPRING

## •Repositorios

```
public interface IFamiliaRepository extends JpaRepository<Familia, Integer> {  
}
```

## •Servicios

```
@Service  
public class FamiliaService {  
    @Autowired  
    private IFamiliaRepository familiaRepository;  
  
    public List<Familia> getAllFamilias() {  
        return familiaRepository.findAll();  
    }  
  
    public Optional<Familia> getFamiliaById(int id) {  
        Optional<Familia> familia = familiaRepository.findById(id);  
        return familia;  
    }  
  
    public Familia createFamilia(Familia familia) {  
        Familia nuevaFamilia = familiaRepository.save(familia);  
        return nuevaFamilia;  
    }  
  
    public Familia updateFamilia(int id, Familia familia) {  
        Familia updatedAve = familiaRepository.save(familia);  
        return updatedAve;  
    }  
  
    public void deleteFamilia(int id) {  
        Optional<Familia> familiaOpt = familiaRepository.findById(id);  
        familiaRepository.delete(familiaOpt.get());  
    }  
}
```

# INTRODUCCIÓN A SPRING

## .Controlador

```
@RestController
@RequestMapping("/api/familias")
public class FamiliaController {

    @Autowired
    private FamiliaService familiaService;

    @GetMapping
    public List<Familia> getAllFamilias() {
        return familiaService.getAllFamilias();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Familia> getFamiliaById(@PathVariable int id) {
        Optional<Familia> familia = familiaService.getFamiliaById(id);
        return familia.map(ResponseEntity::ok).orElse(ResponseEntity.notFound().build());
    }
}
```

# INTRODUCCIÓN A SPRING

## .Controlador

@PostMapping

```
public ResponseEntity<Familia> createFamilia(@RequestBody Familia familia) {  
    Familia nuevaFamilia = familiaService.createFamilia(familia);  
    return ResponseEntity.ok(nuevaFamilia);  
}
```

@PutMapping("/{id}")

```
public ResponseEntity<Familia> updateFamilia(@PathVariable int id, @RequestBody Familia familia) {  
    Familia updatedFamilia = familiaService.createFamilia(familia);  
    return ResponseEntity.ok(updatedFamilia);  
}
```

@DeleteMapping("/{id}")

```
public ResponseEntity<Void> deleteFamilia(@PathVariable int id) {  
    familiaService.deleteFamilia(id);  
    return ResponseEntity.noContent().build();  
}
```



# INTRODUCCIÓN A SPRING

## • Clase ResponseEntity.

• Es una clase de Spring Web) que se utiliza para representar una respuesta HTTP completa, incluyendo:

• El cuerpo de la respuesta (contenido, como JSON, XML, texto, etc.).

• El código de estado HTTP (como 200 OK, 404 Not Found, 500 Internal Server Error, etc.).

• Los encabezados HTTP (headers, como Content-Type, Location, etc.).

## • Sintaxis básica:

```
return new ResponseEntity<>(body, HttpStatus.OK);
```

## • Sintaxis con encabezados:

```
HttpHeaders headers = new HttpHeaders();  
headers.add("Custom-Header", "value");  
return new ResponseEntity<>(body, headers, HttpStatus.OK);
```

# INTRODUCCIÓN A SPRING

- Clase ResponseEntity.

- Métodos estáticos:

- `return ResponseEntity.ok(entidad);`

- `return ResponseEntity.notFound().build();` → build construye la instancia de retorno

# INTRODUCCIÓN A SPRING



## SPRING DATA REST

# INTRODUCCIÓN A SPRING

- Spring Data REST es un módulo de Spring que expondrá automáticamente una API REST basada en tus repositorios JPA (CrudRepository, JpaRepository, etc.), sin necesidad de escribir controladores manuales.

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-data-rest</artifactId>  
</dependency>
```

- Spring Data REST automáticamente expone los siguientes endpoints sin escribir ningún @RestController:

- GET /productos
- POST /productos
- GET /productos/{id}
- PUT /productos/{id}
- DELETE /productos/{id}

# INTRODUCCIÓN A SPRING

## •Usos:

- Proyectos rápidos donde necesitas una API CRUD básica sin lógica personalizada.
- Herramientas internas para desarrolladores o pruebas de conceptos.
- Combinado con configuraciones de seguridad y exposición controlada.

## •Peligros:

- Poca personalización: Difícil controlar validaciones, formatos de respuesta, lógica de negocio.
- Swagger/OpenAPI no lo maneja bien sin configuración adicional.
- Puede exponer datos sensibles si no se configura bien.