

Desenvolvimento de aplicacíons con Spring MVC nun contorno moderno

Antonio Varela

Temario

- ▶ Introducción a Spring MVC
- ▶ Fundamentos e configuración inicial
- ▶ Creación de controladores y manejo de peticiones
- ▶ Desenvolvimiento de Controladores RESTful
- ▶ Integración con Spring Data, Configuración básica y manejo de datos
- ▶ Implementación de Seguridad con Spring Security, Autenticación y configuración de seguridad básica
- ▶ Conclusiones y Mejores Prácticas

Introducción a Spring MVC

Entendiendo el Ecosistema Java y Spring

Java

- ▶ Lenguaje de programación orientado a objetos.
- ▶ Ampliamente utilizado en aplicaciones empresariales, móviles y sistemas distribuidos.
- ▶ Plataforma: “Escribe una vez, ejecuta en cualquier lugar” (Write Once, Run Anywhere).

Java EE (Jakarta EE)

- ▶ Conjunto de especificaciones para el desarrollo de aplicaciones empresariales.
- ▶ Incluye servicios como servlets, JPA (Java Persistence API) y JAX-RS para REST.
- ▶ Proporciona estándares, pero puede ser pesado y requiere servidores de aplicaciones.

Entendiendo el Ecosistema Java y Spring

Spring Framework

- ▶ Framework modular y flexible para el desarrollo en Java.
- ▶ Resuelve limitaciones de Java EE mediante configuración simplificada e integración más eficiente.
- ▶ Spring MVC: Submódulo para crear aplicaciones web siguiendo el patrón Modelo-Vista-Controlador.

Spring Boot

- ▶ Extensión de Spring Framework que simplifica el desarrollo.
- ▶ Características destacadas:
- ▶ Configuración automática (Auto-Configuration).
- ▶ Servidor embebido (Tomcat, Jetty).
- ▶ Incluye dependencias preconfiguradas con “Spring Boot Starters”.
- ▶ Ideal para entornos modernos y desarrollo rápido.

Evolución de Spring y Spring Boot

Spring Framework

- ▶ **Spring 1.x (2004)**: Introducción del concepto de **Inversión de Control (IoC)** e **Inyección de Dependencias (DI)**, con enfoque en modularidad y desacoplamiento.
- ▶ **Spring 2.x (2006)**: Configuración XML mejorada e introducción de **Spring AOP** (Aspect-Oriented Programming).
- ▶ **Spring 3.x (2009)**: Soporte para configuración basada en **anotaciones**, compatibilidad con Java 5 y Java EE 6.
- ▶ **Spring 4.x (2013)**: Soporte para Java 8, funcionalidades de programación funcional y mejoras en aplicaciones RESTful.
- ▶ **Spring 5.x (2017)**: Introducción del soporte para reactive programming con **Spring WebFlux**, compatible con Java 8-21 y optimización para **microservicios y Kotlin**.

Evolución de Spring y Spring Boot

Spring Framework

- ▶ **Spring 6.x (2022)**: Requiere Java 17 como mínimo, migración del espacio de nombres de javax a jakarta para compatibilidad con **Jakarta EE 9+**, soporte mejorado para **imágenes nativas** con GraalVM y enfoque en aplicaciones **cloud-native**
- ▶ **Spring 7.x (2025)**: Planeado para **noviembre de 2025**, con soporte para **Jakarta EE 11**, alineación con **JDK 25 LTS**, integración con Kotlin 2 y adopción de anotaciones de **null-safety** de **JSpecify**.

Evolución de Spring y Spring Boot

Spring Boot

- ▶ **Spring Boot 1.x (2014)**: Primer lanzamiento estable con configuraciones automáticas y servidores embebidos, ideal para desarrollo rápido.
- ▶ **Spring Boot 2.x (2018)**: Compatible con **Spring 5**, soporte para aplicaciones reactivas con Spring WebFlux, mejora del rendimiento y monitoreo con Spring Boot Actuator.
- ▶ **Spring Boot 3.x (2022)**: Requiere **Java 17** como mínimo, integración con GraalVM para aplicaciones nativas, optimización para la nube y entornos de contenedores como **Docker y Kubernetes**.

Evolución de Spring y Spring Boot

Versión de Spring Framework	Versión de Spring Boot	Versión de Java soportada
Spring Framework 4.x	Spring Boot 1.x	Java 6, 7, 8
Spring Framework 5.x	Spring Boot 2.0.x	Java 8, 9, 10, 11
Spring Framework 5.2.x	Spring Boot 2.2.x - 2.5.x	Java 8, 11, 14, 15
Spring Framework 6.x	Spring Boot 3.x	Java 17+ (requiere mínimo Java 17)
Spring Framework 7.x (Próxima versión)	Spring Boot 4.x (Próxima versión)	Java 17+ (probablemente)

El sistema modular de Spring

- ▶ Spring Framework está **diseñado de manera modular**, lo que significa que puedes utilizar solo los módulos que necesites para tu aplicación, sin tener que cargar toda la complejidad del framework. Este enfoque modular permite que Spring sea **ligero** y **flexible**, adaptándose a diferentes necesidades y tipos de aplicaciones.
- ▶ Los módulos de Spring se dividen en grupos que se enfocan en diferentes aspectos del desarrollo de aplicaciones: desde la **gestión de dependencias**, pasando por la **programación orientada a aspectos (AOP)**, hasta el desarrollo de aplicaciones **web** y **enterprise**.
- ▶ <https://start.spring.io>

El sistema modular de Spring

1. Core Container

- ▶ **spring-core**: El núcleo del framework, que proporciona funcionalidades básicas como la gestión de beans y la Inyección de Dependencias (DI).
- ▶ **spring-beans**: Implementa la infraestructura para la inyección de dependencias (DI), gestionando los objetos (beans).
- ▶ **spring-context**: Extiende spring-beans proporcionando soporte adicional para la configuración de la aplicación y la integración de eventos.
- ▶ **spring-expression**: Proporciona el lenguaje de expresión de Spring (SpEL) para evaluar expresiones dentro de la configuración.

El sistema modular de Spring

2. Spring AOP

- ▶ **spring-aop:** Soporta la **programación orientada a aspectos (AOP)**, permitiendo agregar lógica transversal (como logging, seguridad o transacciones) sin modificar el código de negocio.

3. Spring Data

- ▶ **spring-data:** Proporciona soporte para la integración con bases de datos, facilitando la interacción con diferentes tecnologías de persistencia (JPA, MongoDB, Redis, etc.), incluyendo repositorios y consultas automáticas.

4. Spring Web

- ▶ **spring-web:** Proporciona funcionalidades para aplicaciones web tradicionales, como el manejo de peticiones HTTP, controladores MVC, y la integración con tecnologías como JSP y servlets.
- ▶ **spring-mvc:** Implementa el patrón **Modelo-Vista-Controlador (MVC)**, utilizado para crear aplicaciones web escalables y con separación clara de responsabilidades.

El sistema modular de Spring

5. Spring Security

- ▶ **spring-security:** Proporciona soluciones para la autenticación y autorización en aplicaciones, ayudando a proteger aplicaciones web mediante diversas estrategias de seguridad (roles, permisos, etc.).

6. Spring Boot

- ▶ **spring-boot:** Ofrece una configuración automática y simplificada para aplicaciones Spring, facilitando el desarrollo de aplicaciones stand-alone con servidores embebidos, sin necesidad de configuración compleja.

7. Spring Cloud

- ▶ **spring-cloud:** Proporciona herramientas para construir aplicaciones **cloud-native** y microservicios, como descubrimiento de servicios, configuración distribuida, y patrones de resiliencia.

Enfoque cloud-native

- ▶ Filosofía de diseño y desarrollo de software que aprovecha al máximo las capacidades y características de los entornos de computación en la nube.
- ▶ Busca construir aplicaciones que sean **escalables, resilientes, portables y adaptadas** a infraestructuras modernas como **contenedores, Kubernetes y servicios cloud**.

Características cloud-native

1. Microservicios:

- ▶ Las aplicaciones se dividen en servicios independientes, pequeños y especializados que pueden desarrollarse, implementarse y escalarse de forma autónoma. Esto reduce la complejidad y mejora la agilidad en el desarrollo.

2. Contenedores:

- ▶ Se usan tecnologías como **Docker** para empaquetar aplicaciones y sus dependencias, asegurando portabilidad y consistencia entre entornos (desarrollo, pruebas y producción).

3. Orquestación y escalado dinámico:

- ▶ Herramientas como **Kubernetes** gestionan el despliegue, escalado y recuperación automática de contenedores, asegurando alta disponibilidad y eficiencia.

4. Despliegue continuo (CI/CD):

- ▶ Implementación de pipelines de integración y despliegue continuo que permiten actualizar aplicaciones de manera rápida y frecuente, con mínima interrupción.

Características cloud-native

5. Infraestructura como código (IaC):

- ▶ Se automatiza la creación y gestión de infraestructura utilizando herramientas como Terraform o AWS CloudFormation, lo que garantiza consistencia y reduce errores.

6. Desacoplamiento de estado:

- ▶ Las aplicaciones cloud-native tienden a ser **stateless** (sin estado) o a gestionar el estado mediante servicios externos, como bases de datos o cachés distribuidos, para facilitar el escalado horizontal.

7. Resiliencia y tolerancia a fallos:

- ▶ Estas aplicaciones están diseñadas para ser resilientes, usando patrones como reinicio automático, replicación y arquitecturas basadas en eventos para manejar fallos.

8. Optimización para la nube:

- ▶ Aprovechan al máximo los servicios ofrecidos por proveedores de nube pública como AWS, Azure o Google Cloud, tales como bases de datos gestionadas, almacenamiento escalable o servicios serverless.

Beneficios cloud-native

- ▶ **Escalabilidad:** La arquitectura modular y el uso de herramientas como Kubernetes permiten escalar servicios de manera automática según la demanda.
- ▶ **Resiliencia:** Las aplicaciones pueden recuperarse rápidamente de fallos gracias a arquitecturas distribuidas.
- ▶ **Agilidad:** La separación en microservicios y los pipelines CI/CD permiten a los equipos trabajar en paralelo y realizar lanzamientos frecuentes.
- ▶ **Reducción de costos:** Uso eficiente de recursos mediante el pago por consumo y escalado dinámico.
- ▶ **Portabilidad:** Las aplicaciones son independientes del entorno específico y pueden moverse entre nubes públicas, privadas o híbridas.

cloud-native en Spring Framework y Spring boot

- ▶ Spring Framework, especialmente desde **Spring Boot 3** y **Spring Framework 6**, está diseñado para soportar un enfoque cloud-native:
- ▶ • **Spring Boot Actuator**: Ofrece endpoints para monitoreo y gestión, facilitando la integración con herramientas de orquestación como Kubernetes.
- ▶ • **Spring Cloud**: Extensión de Spring para simplificar patrones cloud-native, como descubrimiento de servicios, configuración distribuida, circuit breakers y mensajería.
- ▶ • **Compatibilidad con GraalVM**: Permite construir aplicaciones nativas que se inician más rápido y usan menos recursos, ideales para entornos en la nube.

Patrones de diseño

- ▶ Un **patrón de diseño** es una solución reutilizable y probada para resolver problemas comunes en el desarrollo de software.
- ▶ Estos patrones representan **buenas prácticas** que ayudan a estructurar y organizar el código, promoviendo eficiencia y mantenibilidad.
 1. Resolución de problemas recurrentes
 2. Estandarización del código
 3. Facilitan el mantenimiento y escalabilidad
 4. Promueven la reutilización
 5. Mejores prácticas (Principios SOLID)

Principios SOLID

S: Principio de Responsabilidad Única (SRP):

Cada clase debe tener una única responsabilidad o razón para cambiar.

O: Principio de Abierto/Cerrado (OCP):

El software debe ser **abierto para extensión**, pero **cerrado para modificación**.

Ejemplo: Añadir nuevas reglas de negocio sin cambiar el código base.

L: Principio de Sustitución de Liskov (LSP):

Las subclases deben poder reemplazar a sus clases base sin alterar la funcionalidad.

I: Principio de Segregación de Interfaces (ISP):

Las interfaces deben ser pequeñas y específicas, evitando incluir métodos innecesarios.

D: Principio de Inversión de Dependencias (DIP):

Las clases deben depender de abstracciones, no de implementaciones concretas.

Patrones de diseño en aplicaciones web

- ▶ **Singleton:** Asegura que una clase tenga solo una instancia.
 - ▶ *Ejemplo:* Gestión de la configuración de una aplicación.
- ▶ **Factory Method:** Proporciona una manera de crear objetos sin especificar su clase concreta.
 - ▶ *Ejemplo:* Crear servicios basados en parámetros.
- ▶ **DAO (Data Access Object):** Abstira la interacción con la base de datos.
 - ▶ *Ejemplo:* Acceso a datos con **Spring Data JPA**.
- ▶ **Observer:** Permite notificar automáticamente a varios objetos cuando cambia el estado de uno.
 - ▶ *Ejemplo:* Listeners en eventos de la interfaz gráfica.

Patrones de diseño en aplicaciones web

- ▶ **Patrón Front Controller**
- ▶ **¿Qué hace?**
 - ▶ Centraliza la gestión de las solicitudes entrantes (HTTP) en un único controlador antes de delegarlas a los componentes correspondientes (acciones específicas, vistas, etc.).
- ▶ **Ejemplo en Java:**
 - ▶ Este patrón es la base de frameworks como **Spring MVC**, donde un controlador como DispatcherServlet gestiona todas las solicitudes.

Patrones de diseño en aplicaciones web

Patrón Front Controller

- ▶ Centraliza la gestión de las solicitudes entrantes (HTTP) en un único controlador antes de delegarlas a los componentes correspondientes (acciones específicas, vistas, etc.)

Patrón DAO (Data Access Object)

- ▶ Proporciona una capa de abstracción entre la lógica de negocio y la base de datos, facilitando el acceso a datos de manera uniforme.

Patrón Singleton

- ▶ Garantiza que una clase tenga solo una instancia y proporciona un punto global de acceso a ella.

Patrones de diseño en aplicaciones web

Patrón Builder

- ▶ Facilita la creación de objetos complejos al construirlos paso a paso.

Patrón Proxy

- ▶ Proporciona un sustituto o “apoderado” para controlar el acceso a otro objeto.

Patrón Observer

- ▶ Permite que múltiples objetos (suscriptores) se actualicen automáticamente cuando el estado de otro objeto cambia.

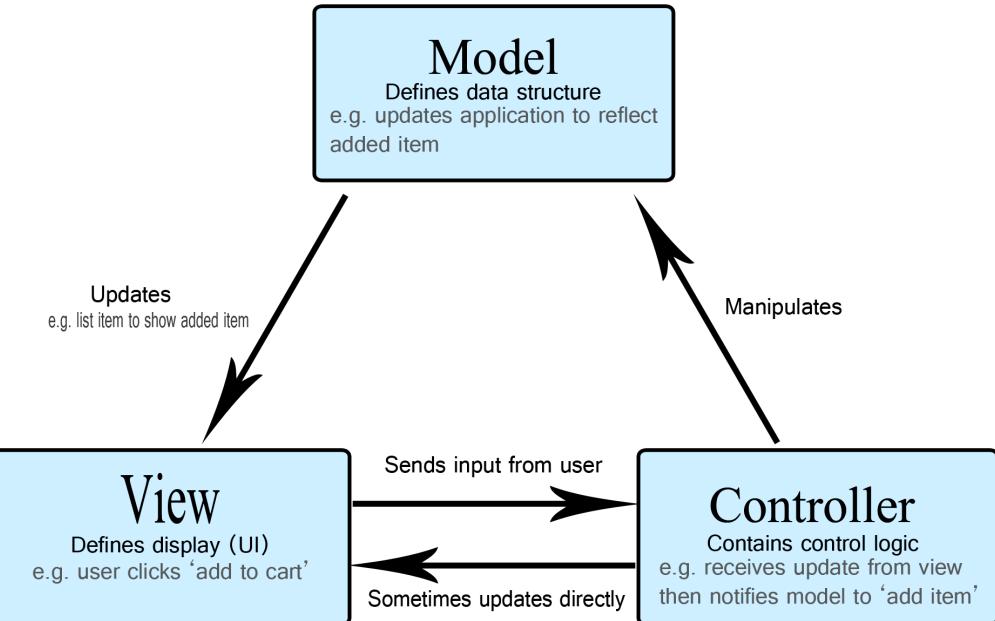
Patrón Factory Method

- ▶ Define una interfaz para crear objetos, pero permite que las subclases decidan qué clase instanciar.

Patrones de diseño en aplicaciones web

Patrón Model-View-Controller (MVC)

- ▶ Divide la aplicación en tres componentes:
 - ▶ **Modelo (Model):** Gestiona los datos y la lógica de negocio.
 - ▶ **Vista (View):** Renderiza la interfaz de usuario.
 - ▶ **Controlador (Controller):** Procesa las solicitudes del usuario y coordina el modelo y la vista.



De MVC a Spring MVC

Spring MVC implementa el patrón MVC de manera robusta y moderna, simplificando el desarrollo de aplicaciones web con características avanzadas:

1. Controladores basados en anotaciones:

- ▶ Los controladores se configuran fácilmente con anotaciones como `@Controller` o `@RestController`.

2. Resolución de peticiones:

- ▶ Se mapean URLs a métodos de controlador mediante `@RequestMapping`.

3. Integración de modelo y vista:

- ▶ Usa objetos como `ModelAndView` para combinar datos (Modelo) con vistas (HTML, JSON).

De MVC a Spring MVC

4. Validación y binding de datos:

- ▶ Permite transformar y validar datos desde la solicitud HTTP al modelo con anotaciones como @Valid.

5. Integración con otras tecnologías:

- ▶ Trabaja fácilmente con Thymeleaf, JSP, o incluso JSON/XML para APIs RESTful

```
@Controller
public class GreetingController {
    @RequestMapping("/greeting")
    public String greeting(Model model) {
        model.addAttribute("message", "¡Hola, Spring MVC!");
        return "greeting"; // Nombre de la vista
    }
}
```

Fundamentos y configuración inicial

¿Qué es Spring MVC?

- ▶ Spring MVC es un módulo del framework Spring diseñado para construir aplicaciones web basadas en el patrón **Modelo-Vista-Controlador (MVC)**.
- ▶ Proporciona herramientas para manejar solicitudes HTTP y generar respuestas dinámicas, ya sea en HTML, JSON, o XML.

Características Clave de Spring MVC

- ▶ **Basado en el patrón MVC:** Facilita la separación de responsabilidades entre la lógica de negocio (Modelo), la lógica de control (Controlador) y la presentación (Vista).
- ▶ **Controladores anotados:** Utiliza anotaciones como `@Controller` o `@RestController` para simplificar la configuración.
- ▶ **Flexible en vistas:** Soporta múltiples tecnologías de vistas, como **Thymeleaf**, **JSP**, o **FreeMarker**.
- ▶ **Enfoque RESTful:** Ideal para construir APIs REST, gracias al soporte de anotaciones como `@RequestMapping` y `@ResponseBody`.
- ▶ **Integración con otras herramientas de Spring:** Como **Spring Security** para la seguridad y **Spring Data** para la persistencia.

Características Clave de Spring MVC

Spring MVC hereda las características fundamentales del **framework Spring**, tales como:

- ▶ **Inyección de dependencias (Dependency Injection):** Simplifica la gestión de componentes y servicios.
- ▶ **Programación Orientada a Aspectos (AOP):** Facilita la implementación de lógica transversal como logging y manejo de excepciones.
- ▶ **Internacionalización:** Soporte nativo para adaptar la aplicación a múltiples idiomas y regiones.
- ▶ **Spring Expression Language (SpEL):** Lenguaje de expresión potente para trabajar con datos y configuraciones.

Características Clave de Spring MVC

Spring MVC introduce características avanzadas diseñadas para simplificar el desarrollo de aplicaciones web basadas en contenedores Java Servlet:

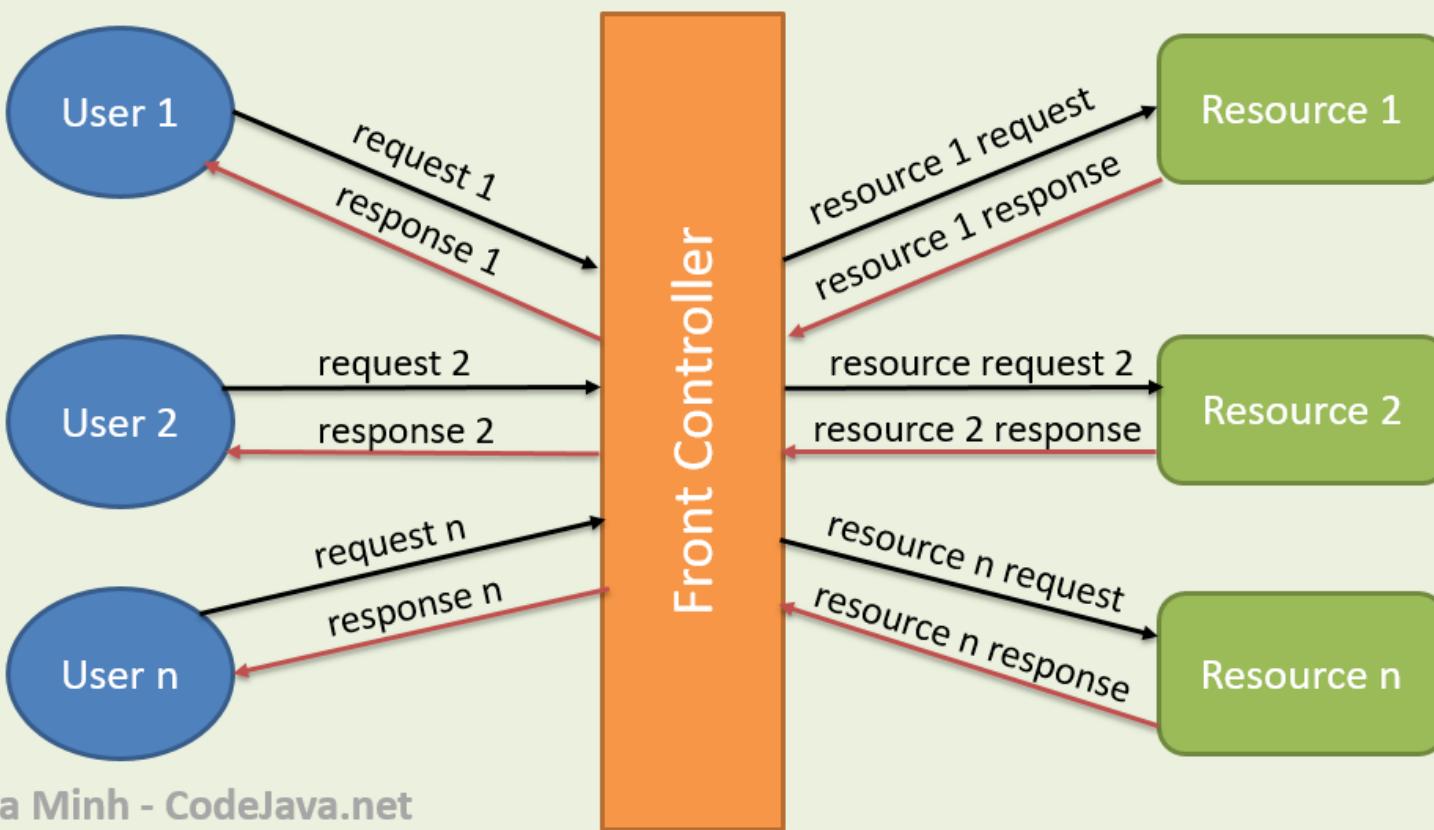
- ▶ **DispatcherServlet:** Actúa como un **Front Controller**, gestionando todas las solicitudes HTTP entrantes y delegándolas a los controladores adecuados.
- ▶ **Manejo de formularios y validación:** Facilita el procesamiento y validación de datos enviados desde formularios web.
- ▶ **Resolución de vistas y más:** Incluye resolutores para vistas, manejo de excepciones, localización, temas, subida de archivos y logging.
- ▶ **Compatibilidad con tecnologías de vista:** Soporte para Thymeleaf, FreeMarker, JSP, JSTL, generación de PDF y Excel.
- ▶ **Integración con Spring Security:** Proporciona autenticación y autorización robustas para aplicaciones web.
- ▶ **Procesamiento asíncrono:** Compatible con el manejo asíncrono de Servlet para mejorar el rendimiento en operaciones de larga duración.

Como funciona Spring MVC

- ▶ Para entender cómo funciona **Spring MVC**, primero es importante familiarizarse con los patrones de diseño **Front Controller** y **Model-View-Controller (MVC)**, ya que son la base filosófica de este framework.
- ▶ El patrón **Front Controller** centraliza la gestión de todas las solicitudes HTTP en un único punto de entrada (como el `DispatcherServlet` en Spring MVC). Esto simplifica el enrutamiento y la delegación hacia los controladores adecuados, promoviendo una arquitectura organizada y fácil de mantener.

Como funciona Spring MVC

Front Controller design pattern

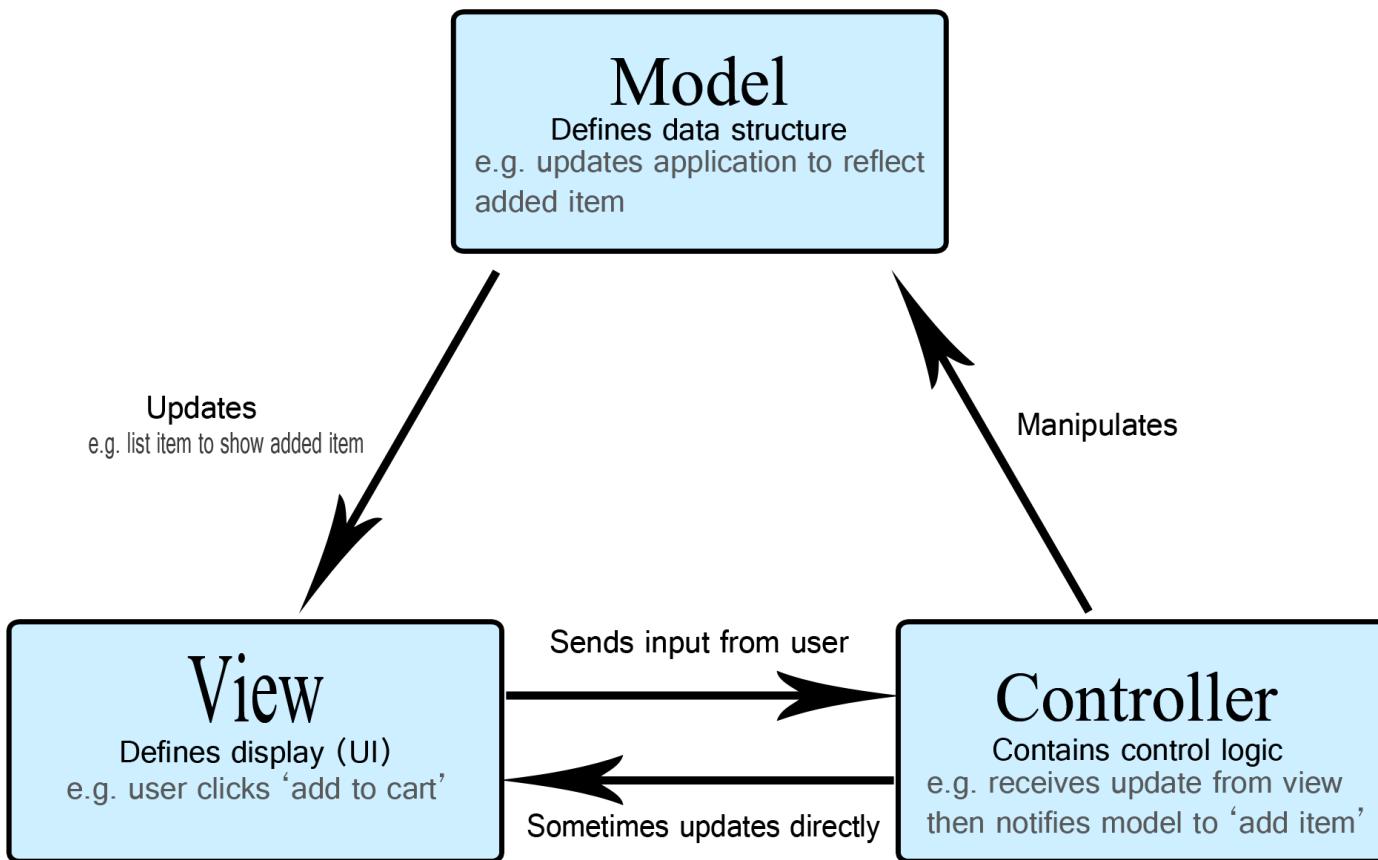


Como funciona Spring MVC

Por otro lado, en el patrón MVC el código de la aplicación se organiza en tres capas lógicas: **Modelo**, **Vista** y **Controlador**, cada una con responsabilidades claramente definidas:

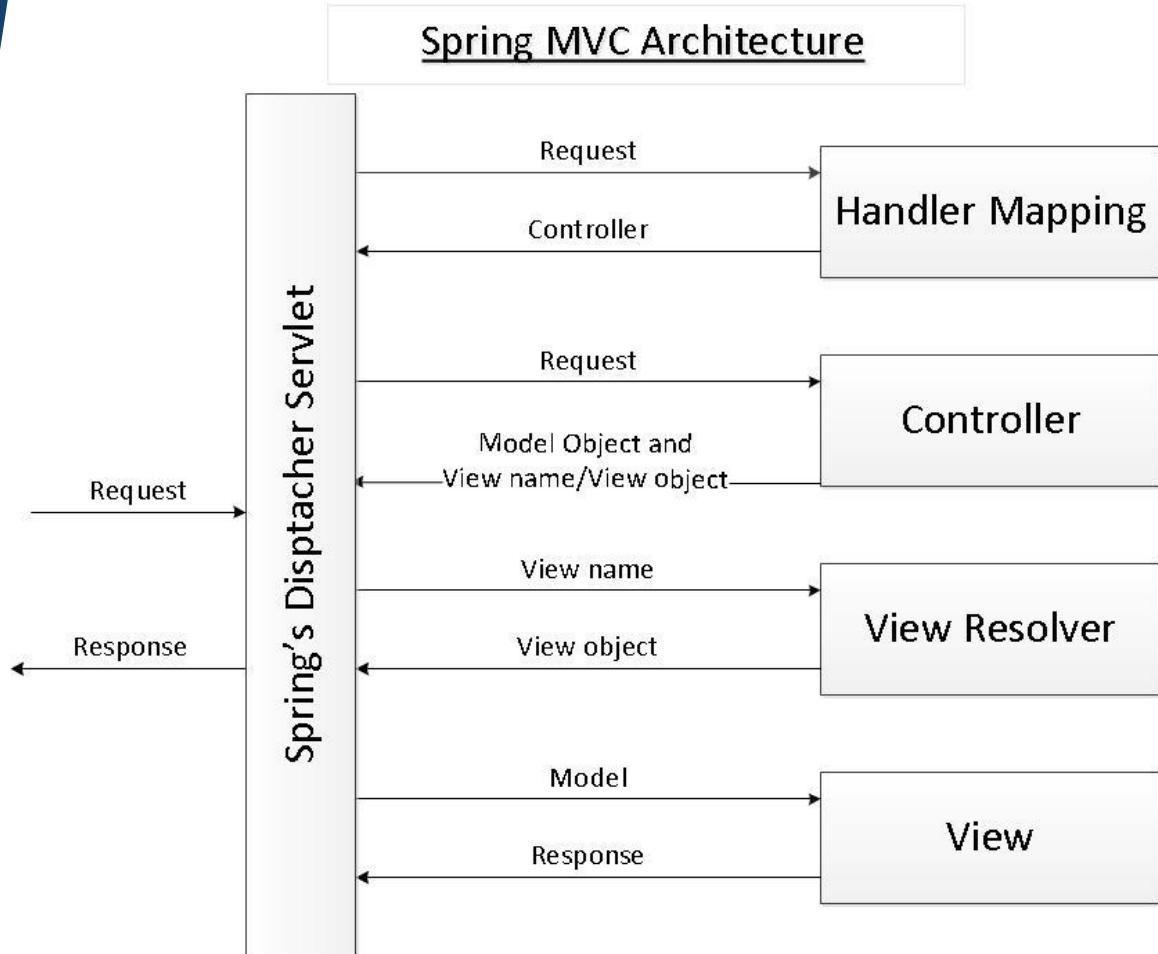
- ▶ • **Modelo (Model)**: Representa los datos de la aplicación y su lógica de negocio. Es responsable de gestionar el estado y las operaciones asociadas a los datos.
- ▶ • **Vista (View)**: Se encarga de renderizar la interfaz de usuario, mostrando los datos del Modelo en un formato comprensible para el usuario.
- ▶ • **Controlador (Controller)**: Maneja las solicitudes entrantes, invoca los servicios de la aplicación que procesan los datos, actualiza el Modelo según sea necesario y selecciona la Vista apropiada para responder a la solicitud.

Como funciona Spring MVC



Como funciona Spring MVC

La combinación de estos dos patrones de diseño aplicados en Spring MVC se traduce en lo siguiente:



Como funciona Spring MVC

- ▶ El **DispatcherServlet** es el componente central de **Spring MVC**, que actúa como un **Front Controller**.
- ▶ Este controlador principal coordina todos los elementos necesarios para procesar una única solicitud HTTP, invocando varios objetos en una secuencia específica:
 1. **Handler Mapping:**
 - ▶ Determina qué URL está asociada con qué método manejador dentro de qué clase de controlador. Esto facilita el enrutamiento de solicitudes a los controladores adecuados.
 2. **Controlador (Controller):**
 - ▶ Recibe la solicitud, analiza los datos de entrada, invoca los servicios o componentes de negocio que procesan esa entrada, y selecciona un nombre lógico para la vista que responderá a la solicitud.

Como funciona Spring MVC

3. View Resolver:

- ▶ Traduce el nombre lógico de la vista seleccionado por el controlador a una plantilla física específica (como un archivo HTML, JSP, Excel, PDF, etc.).

4. Generación de la Respuesta:

- ▶ El controlador selecciona la plantilla de vista apropiada, le pasa los datos del modelo (Model) y genera la respuesta final que se envía al cliente.

Requisitos previos

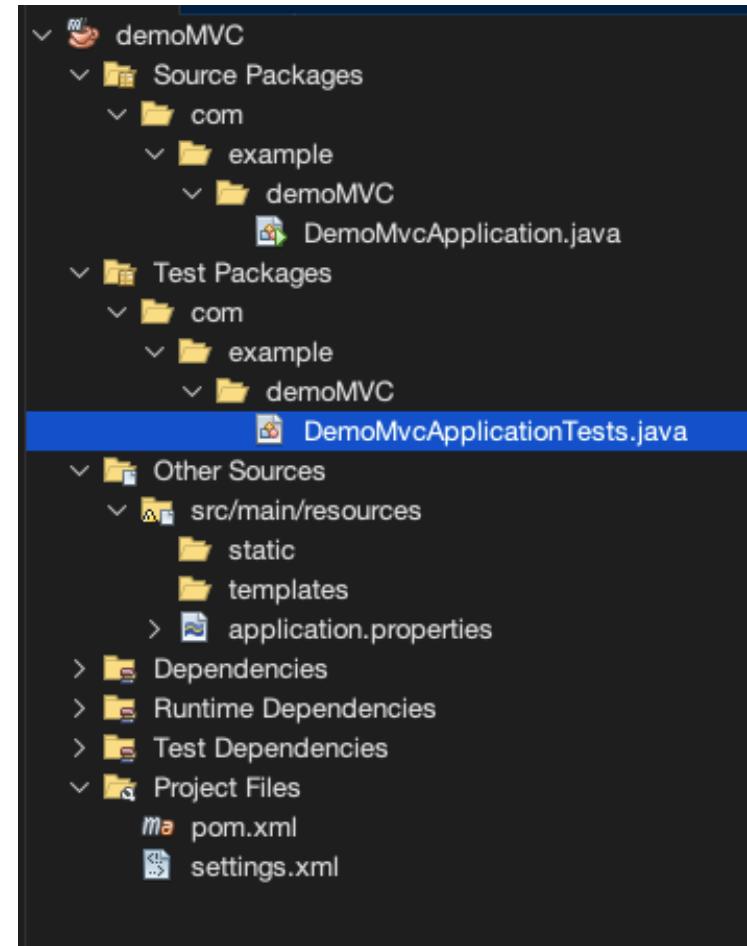
- ▶ **Conocimientos básicos de Java.**
- ▶ Familiaridad con **Maven** o **Gradle** como herramientas de construcción.
- ▶ **Entorno de desarrollo:** IntelliJ IDEA, Eclipse, Netbeans o cualquier otro IDE.
- ▶ Tener configurado:
 - ▶ **Java JDK 17** o superior
 - ▶ **Maven** o **Gradle** instalados

Ejemplo Hello world

1. Inicializamos nuestro proyecto desde <https://start.spring.io>
 - ▶ Elige entre **Gradle** o **Maven** y el lenguaje que deseas usar. Esta guía asume que elegiste **Java**.
 - ▶ Haz clic en **Dependencies** y selecciona **Spring Web**, **Thymeleaf** y **Spring Boot DevTools**.
 - ▶ Haz clic en **Generate**.
 - ▶ Descarga el archivo ZIP resultante, que contiene un proyecto de aplicación web configurado con las opciones que seleccionaste.

Ejemplo Hello world

- ▶ El proyecto descargado tiene la siguiente estructura:
 - ▶ Una clase de aplicación **@SpringBootApplication**
 - ▶ Su clase homónima para test
 - ▶ Un fichero general de propiedades `application.properties`
 - ▶ Los directorios `static` y `templates` para los ficheros de las vistas
 - ▶ Y las dependencias mínimas en el `pom.xml`



Ejemplo Hello world

```
package com.example.demoMVC;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoMvcApplication {

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

Ejemplo Hello world

- ▶ La clase anotada con `@SpringBootApplication` es **el punto de entrada principal** en una aplicación Spring Boot. Esta anotación combina varias características y configuraciones esenciales que facilitan el inicio y la configuración automática de una aplicación
- ▶ `@SpringBootApplication` es una **meta-anotación**, lo que significa que combina varias anotaciones en una sola para simplificar la configuración de una aplicación Spring Boot

1. `@SpringBootConfiguration`:

- ▶ Marca la clase como una fuente de configuración para Spring, equivalente a `@Configuration` en aplicaciones estándar de Spring Framework.

Ejemplo Hello world

2. @EnableAutoConfiguration:

- ▶ Habilita la configuración automática de Spring Boot, encargándose de configurar componentes automáticamente en función de las dependencias declaradas en el proyecto (por ejemplo, si tienes `spring-boot-starter-web`, se configura un servidor embebido como Tomcat o Jetty).

3. @ComponentScan:

- ▶ Escanea automáticamente el paquete base y los subpaquetes en busca de componentes anotados con `@Component`, `@Service`, `@Repository`, `@Controller`, o cualquier otra anotación de estereotipo. Esto elimina la necesidad de especificar explícitamente el paquete de componentes.

Ejemplo Hello world

2. Para este primer ejemplo podemos crear una página index.html estática en el directorio src/main/resources/static

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

</body>
</html>
```

3. Ahora podemos desplegar el proyecto y navegar a <http://localhost:8080>

Creación de controladores y manejo de peticiones

Los controladores

¿Qué es un @Controller en Spring MVC?

- ▶ En **Spring MVC**, la anotación `@Controller` se utiliza para marcar una clase como un **controlador**, que es un componente encargado de manejar las solicitudes HTTP y proporcionar respuestas apropiadas en una aplicación web.
- ▶ El controlador es una parte clave del patrón **Modelo-Vista-Controlador (MVC)**, donde gestiona la lógica de la aplicación relacionada con las solicitudes de los usuarios.

Los controladores

Responsabilidades de un Controlador

- ▶ **Gestionar solicitudes HTTP:**
 - ▶ Un controlador maneja solicitudes HTTP entrantes, las procesa y decide qué acción tomar en respuesta.
- ▶ **Interactuar con el modelo de datos:**
 - ▶ El controlador puede interactuar con los servicios y objetos de negocio para procesar datos y preparar la respuesta.
- ▶ **Seleccionar una vista:**
 - ▶ Despues de procesar los datos, el controlador selecciona una vista que será renderizada para el usuario, pasando los datos necesarios para esa vista.

Los controladores

Ejemplo de un @Controller básico

```
@Controller  
public class GreetingController {  
  
    @RequestMapping("/greeting")  
    public String greeting(Model model) {  
        model.addAttribute("message", "¡Hola desde Spring MVC!");  
        return "greeting"; // Nombre de la vista que se renderiza  
    }  
}
```

Los controladores

Ejemplo de un @Controller básico

- ▶ `@Controller`: Marca la clase como un controlador, lo que indica que esta clase manejará solicitudes HTTP.
- ▶ `@RequestMapping("/greeting")`: Asocia el método `greeting()` con la URL `/greeting`. Esta anotación puede ser utilizada para especificar diferentes métodos HTTP, como GET, POST, etc.
- ▶ `Model`: Se utiliza para pasar datos del controlador a la vista. En este caso, se añade un atributo `message` con un valor.
- ▶ `return "greeting"`: El método devuelve el nombre de la vista que se debe mostrar al usuario. Spring buscará un archivo con ese nombre (por ejemplo, `greeting.html` en el caso de Thymeleaf).

Los controladores

Puntos Clave sobre @Controller:

- ▶ **Manejo de rutas:** Cada método dentro del controlador puede estar asociado con una ruta específica usando anotaciones como `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc.
- ▶ **Integración con vistas:** El controlador generalmente selecciona una vista para devolver al usuario, que puede ser un archivo HTML, JSP, o incluso un formato como JSON si se está construyendo una API REST.
- ▶ **No devuelve directamente respuestas:** Aunque el controlador maneja la lógica de la solicitud, no devuelve directamente una respuesta HTTP. En lugar de eso, selecciona una vista y proporciona los datos necesarios.

Los controladores

Diferencia entre `@Controller` y `@RestController`

- ▶ `@Controller`: Utilizado principalmente para aplicaciones web tradicionales donde la vista es renderizada (HTML, JSP, etc.). Los métodos del controlador devuelven el nombre de la vista que debe ser renderizada.
- ▶ `@RestController`: Similar a `@Controller`, pero se utiliza para servicios web RESTful. Los métodos en un `@RestController` devuelven datos directamente (generalmente en formato JSON o XML), sin necesidad de vistas.

Las vistas

- ▶ En el ejemplo del controlador anterior, el método greeting() devuelve el nombre de una vista llamada greeting, que Spring MVC asocia con un archivo de plantilla llamado greeting.html.
- ▶ Este archivo representa la **interfaz de usuario** que se mostrará en el navegador cuando un usuario acceda a la ruta /greeting.

Las vistas

¿Cómo funcionan las vistas en Spring MVC?

1. Motor de Plantillas:

- ▶ Spring MVC soporta varios motores de plantillas para renderizar vistas, como:
 - ▶ **Thymeleaf (recomendado)**: Popular para HTML dinámico con integración completa con Spring.
 - ▶ JSP (Java Server Pages): Una opción más tradicional.
 - ▶ FreeMarker o Velocity: Motores alternativos para generar contenido dinámico.

2. Ubicación de las vistas:

- ▶ Por defecto, las vistas deben ubicarse en el directorio:
 - ▶ src/main/resources/templates/ (para motores como Thymeleaf).

3. Datos del Modelo:

- ▶ Los datos añadidos al modelo en el controlador (model.addAttribute) se pasan automáticamente a la vista y pueden utilizarse en la plantilla para generar contenido dinámico.

Las vistas

Ejemplo de greeting.html utilizando Thymeleaf

src/main/resources/templates/greeting.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Saludo</title>
</head>
<body>
    <h1 th:text="'Mensaje: ' + ${message}"></h1>
    <p>Este es un ejemplo de una vista generada con Thymeleaf.</p>
</body>
</html>
```

Las vistas

Explicación de greeting.html utilizando Thymeleaf

1. Thymeleaf Namespace (`xmlns:th`):

- ▶ Declara que el archivo usará las etiquetas y atributos específicos de Thymeleaf para procesar contenido dinámico.

2. `th:text`:

- ▶ Este atributo reemplaza el contenido del elemento con el valor dinámico del modelo. En este caso, se utiliza la variable `message` que fue añadida al modelo en el controlador:

```
model.addAttribute("message", "¡Hola desde Spring MVC!");
```

Las vistas

Explicación de greeting.html utilizando Thymeleaf

3. Contenido Estático y Dinámico:

- ▶ El texto dentro del párrafo `<p>` es estático y siempre será igual.
- ▶ La etiqueta `<h1>` usa Thymeleaf para mostrar contenido dinámico basado en los datos pasados desde el controlador.

Flujo de invocación

¿Qué ocurre cuando se accede a /greeting?

1. El usuario accede a la URL /greeting.
2. El **DispatcherServlet** identifica el controlador correspondiente (GreetingController) gracias al mapeo definido con @RequestMapping ("/greeting").
3. El controlador procesa la solicitud, añade un mensaje al modelo (message) y selecciona la vista lógica greeting.
4. Spring MVC utiliza el **View Resolver** para encontrar el archivo físico de la vista (greeting.html en el directorio de plantillas).
5. El motor Thymeleaf procesa la plantilla greeting.html, reemplaza las variables dinámicas con los datos del modelo y genera una página HTML completa.
6. La página resultante se envía al navegador del usuario.

Parámetros de invocación en Spring MVC

Parámetros en Solicituds GET

- ▶ En una solicitud **GET**, los parámetros se envían como parte de la URL, generalmente en la forma:
 - ▶ <http://example.com/path?param1=value1¶m2=value2>

```
@Controller
public class GreetingController {

    @GetMapping("/greeting")
    public String greeting(@RequestParam(name = "name", defaultValue = "Mundo") String name, Model model) {
        model.addAttribute("message", "¡Hola, " + name + "!");
        return "greeting"; // Vista que se mostrará
    }
}
```

Parámetros de invocación en Spring MVC

Parámetros en Solicituds GET

- ▶ **@RequestParam:**
 - ▶ Captura el parámetro name enviado en la URL (?name=valor).
 - ▶ El atributo defaultValue define un valor por defecto si el parámetro no se proporciona.
- ▶ **URL de Ejemplo:**
 - ▶ /greeting?name=Juan → Muestra: “¡Hola, Juan!”
 - ▶ /greeting (sin parámetro) → Muestra: “¡Hola, Mundo!”

Parámetros de invocación en Spring MVC

Parámetros en Solicitudes POST

- ▶ En una solicitud **POST**, los parámetros se envían en el cuerpo de la solicitud, a menudo a través de formularios HTML

```
@Controller
public class FormController {

    @PostMapping("/submitForm")
    public String handleForm(@RequestParam("username") String username,
                           @RequestParam("email") String email, Model model) {
        model.addAttribute("message", "Usuario: " + username + ", Email: " + email);
        return "formResponse"; // Vista que mostrará los datos enviados
    }
}
```

Parámetros de invocación en Spring MVC

Parámetros en Solicitudes POST

```
<form action="/submitForm" method="post">
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username">

    <label for="email">Email:</label>
    <input type="email" id="email" name="email">

    <button type="submit">Enviar</button>
</form>
```

Parámetros de invocación en Spring MVC

Parámetros en Solicitudes POST

- ▶ **@RequestParam:**
 - ▶ Captura los valores enviados por el formulario (por ejemplo, los campos username y email).
- ▶ **Método POST:**
 - ▶ La solicitud se envía al controlador usando el método **POST**, especificado en el atributo method del formulario.
- ▶ **Vista Resultante:**
 - ▶ La vista formResponse puede mostrar los datos enviados, por ejemplo:
 - ▶ “Usuario: Juan, Email: juan@example.com”

Parámetros GET y POST con Objetos

- ▶ Spring permite capturar múltiples parámetros en una clase modelo, lo que simplifica la gestión de datos complejos.

```
public class User {  
    private String username;  
    private String email;  
  
    // Getters y setters  
}
```

```
@PostMapping("/submitForm")  
public String handleForm(User user, Model model) {  
    model.addAttribute("message", "Usuario: " + user.getUsername() + ", Email: " + user.getEmail());  
    return "formResponse";  
}
```

- ▶ Spring automáticamente enlaza los parámetros del formulario con los campos de la clase User, eliminando la necesidad de manejar cada parámetro individualmente.

Uso de @PathVariable en Spring MVC

- ▶ `@PathVariable` se utiliza para capturar valores dinámicos de la URL en Spring MVC. Es ideal para rutas en las que los parámetros forman parte del **path** en lugar de ser enviados como parámetros de consulta (**query parameters**).

```
@Controller
public class UserController {

    @GetMapping("/users/{id}")
    public String getUserId(@PathVariable("id") Long userId, Model model) {
        // Lógica para obtener el usuario por ID
        model.addAttribute("userId", userId);
        return "userProfile"; // Nombre de la vista
    }
}
```

- ▶ `@PathVariable("id")`: Captura el valor de `{id}` en la URL `/users/{id}`
 - ▶ Ejemplo: `/users/123` → El valor 123 se asigna a `userId`.

Uso de @PathVariable en Spring MVC

Ventajas de @PathVariable

- ▶ Permite construir URLs **legibles y semánticas**.
- ▶ Útil para rutas RESTful (por ejemplo, /users/{id} para obtener un usuario específico).
- ▶ Compatible con tipos primitivos, objetos (Long, String, etc.) y validaciones personalizadas.

Vistas en Spring MVC

¿Qué es una Vista?

Una vista es el componente del patrón MVC encargado de generar la interfaz de usuario, mostrando los datos procesados por el controlador al usuario final.

► Motores de Plantillas Soportados:

- ▶ **Thymeleaf:** El más utilizado en aplicaciones Spring modernas.
- ▶ **JSP:** Tradicional, basado en Java Server Pages.
- ▶ **FreeMarker y Velocity:** Para plantillas más avanzadas.
- ▶ **PDF y Excel:** Generación de documentos específicos.

► Ubicación por Defecto:

- ▶ Las vistas suelen almacenarse en `src/main/resources/templates` (para Thymeleaf y similares).

Thymeleaf - El Motor de Plantillas Recomendado

¿Qué es Thymeleaf?

Un motor de plantillas para generar HTML dinámico, con integración nativa en Spring Boot.

► Características Principales:

- ▶ Soporte para expresiones dinámicas con \${}.
- ▶ Compatibilidad con plantillas HTML estándar.
- ▶ Fácil integración con datos del modelo.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Ejemplo Thymeleaf</title>
</head>
<body>
    <h1 th:text="`Bienvenido, ' + ${username}`"></h1>
</body>
</html>
```

JSP (Java Server Pages)

¿Qué es JSP?

Una tecnología más antigua para generar contenido dinámico en servidores web.

- ▶ **Ubicación de los Archivos JSP:**

- ▶ Generalmente en el directorio src/main/webapp/WEB-INF/jsp/

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Ejemplo JSP</title>
</head>
<body>
    <h1>Bienvenido, ${username}</h1>
</body>
</html>
```

- ▶ JSP no es tan moderno ni intuitivo como Thymeleaf, por lo que su uso ha disminuido.

FreeMarker y Velocity

FreeMarker y Velocity:

- ▶ Motores de plantillas flexibles y ligeros, ideales para aplicaciones avanzadas

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo FreeMarker</title>
</head>
<body>
    <h1>Bienvenido, ${username}</h1>
</body>
</html>
```

- ▶ **Configuración en Spring Boot:**

- ▶ Necesitas incluir la dependencia correspondiente y configurar el prefijo/sufijo en application.properties.

Thymeleaf vs FreeMarker/Velocity

Característica	Thymeleaf	FreeMarker/Velocity
Popularidad y Uso	Muy popular, especialmente en proyectos modernos con Spring Boot.	Menos popular en la actualidad, aunque aún se utiliza en algunos casos.
Facilidad de Integración	Integración nativa y fluida con Spring Boot.	Requiere configuración adicional para integrarse con Spring.
Compatibilidad con HTML	Diseñado para plantillas HTML naturales, lo que permite trabajar con HTML válido directamente en el navegador.	No está diseñado específicamente para HTML, aunque puede usarse para ello.
Soporte Dinámico	Usa expresiones estándar como \${variable} y soporta operaciones dinámicas avanzadas.	Similar, pero menos intuitivo en algunos casos.
Dependencias	Incluido de forma predeterminada en muchos proyectos Spring Boot.	Requiere dependencias externas específicas.
Curva de Aprendizaje	Fácil de aprender, especialmente para desarrolladores familiarizados con Spring.	Más complejo, especialmente para principiantes.
Uso en Aplicaciones Modernas	Ideal para aplicaciones web modernas con diseño adaptado y dinámico.	Más orientado a proyectos antiguos o necesidades específicas de plantillas ligeras.
Manejo de Datos	Integra perfectamente con el modelo Spring MVC, permitiendo inyección directa de datos en la plantilla.	Puede hacer lo mismo, pero con más trabajo de configuración.
Ecosistema y Documentación	Amplia documentación y ejemplos actualizados.	Documentación suficiente, pero con menor soporte en proyectos actuales.
Compatibilidad con Spring Boot	Excelente. Funciona perfectamente con los ViewResolvers de Spring.	Compatible, pero necesita configuración adicional.

Generación de Documentos PDF y Excel

Spring MVC soporta la generación de documentos PDF y Excel como vistas.

▶ **Ejemplo de Configuración:**

- ▶ Usa bibliotecas como **Apache POI** (para Excel) o **iText** (para PDF).
- ▶ Implementa clases personalizadas que extienden `AbstractPdfView` o `AbstractExcelView`.

Resumen de tecnologías de vistas

- ▶ **Thymeleaf:** La opción preferida para aplicaciones web modernas.
- ▶ **JSP:** Úsalo solo si trabajas en proyectos heredados.
- ▶ **FreeMarker/Velocity:** Útiles para plantillas ligeras y personalizadas.
- ▶ **PDF/Excel:** Para necesidades específicas de generación de documentos.

- ▶ En general, **Thymeleaf** es la mejor opción para nuevos proyectos Spring Boot debido a su simplicidad, compatibilidad nativa y enfoque en HTML moderno. Por otro lado, **FreeMarker** o **Velocity** pueden ser útiles para proyectos heredados o casos de uso específicos.

Elementos Condicionales en Thymeleaf

Thymeleaf permite mostrar u ocultar elementos HTML basados en condiciones.

```
<p th:if="${user.isLoggedIn()}">Bienvenido, <span th:text="${user.name}">Usuario</span></p>
<p th:unless="${user.isLoggedIn()}">Por favor, inicie sesión.</p>
```

- ▶ **th:if:** Muestra el elemento si la condición es verdadera.
- ▶ **th:unless:** Muestra el elemento si la condición es falsa.

Bucles con Thymeleaf

Thymeleaf permite iterar sobre listas o arrays dinámicamente usando th:each.

```
<ul>
    <li th:each="product : ${products}" th:text="${product.name}"></li>
</ul>
```

- ▶ **th:each:**
 - ▶ Itera sobre la colección \${products}.
 - ▶ La variable product contiene el elemento actual en cada iteración.

Uso de atributos dinámicos

Los valores de atributos HTML pueden generarse dinámicamente usando th:attr.

```
<a th:href="@{/products/{id}(id=${product.id})}" th:text="${product.name}"></a>
```

- ▶ th:href: Genera un atributo href dinámico.
 - ▶ Si product.id = 42 y product.name = "Televisor", el enlace generado será:
 - ▶ Televisor

Manejo de Fragmentos con Thymeleaf

Los fragmentos permiten reutilizar partes de la plantilla, como menús o cabeceras

- ▶ Fragmento Reutilizable (header.html):

```
<div th:fragment="header">
    <h1>Mi Tienda Online</h1>
    <p>Bienvenido a nuestra tienda.</p>
</div>
```

Manejo de Fragmentos con Thymeleaf

- ▶ **Uso del Fragmento (index.html):**

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <div th:replace="header :: header"></div>
    <p>Contenido principal aquí.</p>
</body>
</html>
```

- ▶ El contenido del fragmento header se insertará dinámicamente en index.html.

Mensajes Internationalizados (i18n)

Thymeleaf permite mostrar mensajes traducidos usando messages.properties

- ▶ Archivo messages.properties:

```
welcome.message=¡Bienvenido, {0}!
```

- ▶ Vista Thymeleaf:

```
<p th:text="#{welcome.message(${user.name})}"></p>
```

- ▶ Si user.name = "Juan", el resultado será:
 - ▶ "¡Bienvenido, Juan!"

Desenvolvimento de Controladores RESTful

¿Qué es RESTful?

RESTful se refiere a un estilo arquitectónico basado en los principios de **REST (Representational State Transfer)**, diseñado para sistemas distribuidos como servicios web.

Características Clave:

- ▶ **Basado en Recursos:** Cada recurso (como un usuario o un producto) tiene una URL única.
- ▶ Ejemplo: /api/users/1 representa al usuario con ID 1.
- ▶ **Operaciones mediante Métodos HTTP:** Realiza operaciones CRUD (Crear, Leer, Actualizar, Eliminar) utilizando métodos HTTP estándar.
- ▶ **Intercambio de Datos en Formatos Comunes:** JSON es el formato más utilizado, aunque también se soportan XML, texto plano, etc.

Métodos HTTP Principales en RESTful

Método HTTP	Propósito	Ejemplo RESTful
GET	Recuperar datos o recursos.	GET /api/users (obtener lista de usuarios)
POST	Crear un nuevo recurso.	POST /api/users (crear un usuario nuevo)
PUT	Actualizar un recurso existente, reemplazándolo completamente.	PUT /api/users/1 (actualizar al usuario con ID 1)
PATCH	Actualizar parcialmente un recurso existente.	PATCH /api/users/1 (actualización parcial del usuario)
DELETE	Eliminar un recurso.	DELETE /api/users/1 (eliminar usuario con ID 1)

@RestController

- ▶ Es una anotación de Spring que combina las funcionalidades de **@Controller** y **@ResponseBody**.
- ▶ Se utiliza para construir APIs RESTful que devuelven datos directamente en formatos como JSON o XML.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return new User(id, "Juan", "juan@example.com");
    }
}
```

@RestController

Explicación:

- ▶ `@RestController`: Define un controlador RESTful.
- ▶ `@RequestMapping`: Define el prefijo de URL.
- ▶ `@GetMapping("/{id}")`: Captura el ID del usuario desde la URL.
- ▶ Devuelve un objeto `User` que Spring convierte automáticamente en JSON.

Recibiendo Datos en JSON

- ▶ Muchas veces los Rest Controller reciben datos en formato JSON

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public String createUser(@RequestBody User user) {
        return "Usuario creado: " + user.getName();
    }
}
```

- ▶ **@RequestBody:**
 - ▶ Mapea automáticamente el JSON recibido en el cuerpo de la solicitud a un objeto Java.

Generando JSON como respuesta

- ▶ Los controladores RESTful generan automáticamente JSON como salida cuando devuelven objetos Java.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return new User(id, "Juan", "juan@example.com");
    }
}
```

Validación de Entradas con @Valid

Spring puede validar automáticamente datos JSON utilizando anotaciones como @Valid.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public String createUser(@Valid @RequestBody User user) {
        return "Usuario creado: " + user.getName();
    }
}
```

Validación de Entradas con @Valid

Modelo con Anotaciones de Validación:

```
public class User {  
  
    @NotNull  
    private Long id;  
  
    @NotEmpty  
    private String name;  
  
    @Email  
    private String email;  
  
    // Getters y Setters  
}
```

Validación de Entradas con @Valid

- ▶ Entrada válida

```
{  
    "id": 1,  
    "name": "Juan",  
    "email": "juan@example.com"  
}
```

- ▶ Entrada inválida

```
{  
    "id": 1,  
    "name": "",  
    "email": "juan.com"  
}
```

- ▶ Respuesta: **400 Bad Request** con detalles del error.

¿Qué es OpenAPI y Swagger UI?

OpenAPI Specification (OAS):

- ▶ Un estándar para describir, producir, consumir y visualizar APIs RESTful. Permite generar documentación estructurada que es comprensible tanto para humanos como para máquinas.

Swagger UI:

- ▶ Una herramienta visual e interactiva que utiliza la especificación OpenAPI para:
 - ▶ Mostrar la documentación de las APIs en un navegador web.
 - ▶ Probar endpoints directamente desde la interfaz.
 - ▶ Facilitar la comunicación entre equipos (desarrolladores, testers y clientes).

¿Qué es OpenAPI y Swagger UI?

Beneficios Clave:

- ▶ Simplifica el desarrollo de APIs RESTful.
- ▶ Permite probar y depurar endpoints rápidamente.
- ▶ Mejora la comprensión y el consumo de las APIs por terceros.

Ejemplo de URL de Swagger UI:

- ▶ <http://localhost:8080/swagger-ui.html>

¿Qué es OpenAPI y Swagger UI?

Añadir Swagger UI a un Proyecto Spring Boot

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.2.0</version>
</dependency>
```

- ▶ En la mayoría de los casos, no necesitas configuraciones adicionales. **Springdoc** detectará automáticamente tus controladores REST y generará la documentación de la API.

¿Qué es OpenAPI y Swagger UI?

(Opcional) Personalización de Swagger

- ▶ Si deseas personalizar el título, descripción o versión de tu documentación, puedes configurar un archivo application.properties o application.yml:

```
springdoc.api-docs.path=/api-docs  
springdoc.swagger-ui.path=/swagger-ui
```

¿Qué es OpenAPI y Swagger UI?

(Opcional) Anotaciones para Enriquecer la Documentación

- ▶ Puedes usar anotaciones para proporcionar más información sobre tus endpoints REST:
 - ▶ `@Operation`: Describe un método específico.
 - ▶ `@Parameter`: Describe parámetros de entrada.
 - ▶ `@ApiResponse`: Describe respuestas posibles.

```
@RestController
public class ExampleController {

    @Operation(summary = "Obtiene un mensaje de saludo", description = "Endpoint para saludar")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Operación exitosa"),
        @ApiResponse(responseCode = "404", description = "Usuario no encontrado")
    })
    @GetMapping("/hello/{name}")
    public String sayHello(
        @Parameter(description = "Nombre del usuario") @PathVariable String name) {
        return "Hola, " + name + "!";
    }
}
```

Resumen RESTful

- ▶ RESTful facilita la comunicación entre aplicaciones a través de HTTP.
- ▶ Utiliza `@RestController` para construir APIs RESTful de forma simple.
- ▶ Los métodos HTTP (GET, POST, PUT, DELETE, etc.) se usan para mapear operaciones CRUD.
- ▶ JSON es el formato estándar para intercambiar datos.
- ▶ Usa anotaciones como `@RequestBody` para recibir JSON y `@Valid` para validar datos.

Integración con Spring Data

¿Qué es la persistencia?

La **persistencia** se refiere al proceso de almacenar datos de una aplicación en un medio no volátil (como una base de datos) para que puedan ser recuperados y utilizados posteriormente.

Objetivo:

- ▶ Asegurar que los datos sobrevivan más allá de la ejecución de la aplicación.

Características Principales:

- ▶ Almacenamiento de datos estructurados o no estructurados.
- ▶ Recuperación eficiente de los datos.
- ▶ Consistencia y seguridad de la información.

Opciones de persistencia

Bases de Datos Relacionales (RDBMS):

- ▶ **Ejemplos:** MySQL, PostgreSQL, Oracle, SQL Server.
- ▶ Organizan los datos en tablas, filas y columnas.
- ▶ Usan **SQL** para manipulación de datos.
- ▶ Ideal para datos estructurados y relaciones complejas.

Opciones de persistencia

Bases de Datos No Relacionales (NoSQL):

- ▶ **Ejemplos:** MongoDB, Cassandra, Redis, Elasticsearch.
- ▶ Diseñadas para manejar datos no estructurados, semiestructurados o grandes volúmenes de datos.
- ▶ Modelos flexibles como documentos, clave-valor, gráficos, etc.

Opciones de persistencia

Persistencia en Memoria:

- ▶ **Ejemplos:** Redis, H2 (embedded database).
- ▶ Almacena datos temporalmente en la RAM.
- ▶ Utilizado para almacenamiento rápido o pruebas.

ORM (Object Relational Mapping):

- ▶ **Ejemplos:** Hibernate, JPA (Java Persistence API).
- ▶ Facilita la persistencia al mapear objetos Java a tablas de bases de datos.

¿Qué es Spring Data?

Spring Data es un proyecto de Spring que simplifica el acceso y la manipulación de datos persistentes mediante una abstracción uniforme, sin importar la base de datos que se utilice.

Objetivos de Spring Data:

- ▶ Reducir la complejidad al trabajar con bases de datos.
- ▶ Proveer implementaciones genéricas para operaciones comunes (CRUD).
- ▶ Ofrecer integración fluida con **JPA**, bases de datos NoSQL, y otras tecnologías.

¿Qué es Spring Data?

Componentes Principales:

- ▶ **Spring Data JPA:** Soporte para bases de datos relacionales utilizando JPA/Hibernate.
- ▶ **Spring Data MongoDB:** Persistencia con bases de datos NoSQL como MongoDB.
- ▶ **Spring Data Redis, Cassandra, Elasticsearch, etc.:** Opciones especializadas para otras tecnologías de bases de datos.

Ventajas:

- ▶ Reduce el código repetitivo al manejar datos.
- ▶ Compatible con repositorios y consultas personalizadas.
- ▶ Fácil integración con Spring Boot y Spring MVC.

¿Cómo Integrar Spring Data con Spring MVC?

Paso 1: Agregar las Dependencias al Proyecto

- ▶ Ejemplo para bases de datos relacionales usando JPA:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

¿Cómo Integrar Spring Data con Spring MVC?

Paso 2: Configurar la Base de Datos en application.properties

- ▶ Configura los detalles de conexión a MySQL en el archivo src/main/resources/application.properties:

```
# Configuración de la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/nombre_base_datos
spring.datasource.username=tu_usuario
spring.datasource.password=tu_contraseña

# Configuración de JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

¿Cómo Integrar Spring Data con Spring MVC?

- ▶ **spring.datasource.url:** La URL de conexión incluye el host, puerto y el nombre de la base de datos (en este caso, nombre_base_datos).
 - ▶ Ejemplo: jdbc:mysql://localhost:3306/mi_base_de_datos.
- ▶ **spring.datasource.username** y **spring.datasource.password:** Usuario y contraseña de la base de datos MySQL.
- ▶ **spring.jpa.hibernate.ddl-auto:** Controla cómo Hibernate maneja la estructura de la base de datos.
 - ▶ update: Actualiza la estructura existente.
 - ▶ create: Crea la base de datos desde cero en cada inicio.
 - ▶ validate: Solo valida la estructura sin modificarla.
- ▶ **spring.jpa.show-sql:** Muestra las consultas SQL ejecutadas en la consola.
- ▶ **spring.jpa.properties.hibernate.dialect:** Indica el dialecto específico para MySQL.

¿Cómo Integrar Spring Data con Spring MVC?

Paso 3: Crear la Entidad

```
import jakarta.persistence.*;  
  
@Entity  
public class User {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String name;  
    private String email;  
  
    // Getters y Setters
```

¿Cómo Integrar Spring Data con Spring MVC?

Paso 4: Crear el Repositorio

Define una interfaz que extienda de JpaRepository para realizar operaciones CRUD automáticamente:

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // Consultas personalizadas opcionales
    User findByEmail(String email);
}
```

¿Cómo Integrar Spring Data con Spring MVC?

Paso 5: Crear un Controlador REST

Implementa un controlador para interactuar con los datos de la base de datos haciendo uso del repositorio:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    // Obtener todos los usuarios
    @GetMapping
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
}
```

¿Cómo Integrar Spring Data con Spring MVC?

Paso 6: Crear la Base de Datos en MySQL

- ▶ Asegúrate de usar el nombre de la base de datos configurado en application.properties

Paso 7: Iniciar la Aplicación

- ▶ Ejecuta tu aplicación Spring Boot. Spring creará automáticamente las tablas en la base de datos (según la configuración spring.jpa.hibernate.ddl-auto=update).

Implementación de Seguridad con Spring Security

Importancia de la Seguridad en Proyectos Web

Amenazas Comunes en Aplicaciones Web:

- ▶ Robo de datos mediante ataques como **SQL Injection** o **XSS (Cross-Site Scripting)**.
- ▶ Acceso no autorizado a recursos o datos sensibles.
- ▶ Ataques de fuerza bruta para descifrar contraseñas.
- ▶ **CSRF (Cross-Site Request Forgery)**: Solicitudes malintencionadas disfrazadas de solicitudes válidas.

Importancia de la Seguridad en Proyectos Web

Objetivos de la Seguridad:

- ▶ **Confidencialidad:** Garantizar que los datos solo sean accesibles por usuarios autorizados.
- ▶ **Integridad:** Proteger los datos contra modificaciones no autorizadas.
- ▶ **Disponibilidad:** Asegurar que los servicios estén disponibles y protegidos contra ataques.

Por qué es esencial:

- ▶ Incrementar la confianza de los usuarios.
- ▶ Cumplir con normativas y regulaciones (como GDPR).
- ▶ Proteger la reputación del sistema y del negocio.

¿Qué es Spring Security?

Spring Security es un **framework de seguridad** que proporciona autenticación, autorización y protección contra vulnerabilidades comunes en aplicaciones Java/Spring.

Rol en los Proyectos Web:

- ▶ Manejo seguro de sesiones y autenticación de usuarios.
- ▶ Control de acceso a recursos mediante políticas y roles.
- ▶ Protección contra amenazas como ataques CSRF, XSS, etc.

Integración Natural con Spring:

- ▶ Se integra perfectamente con aplicaciones desarrolladas con **Spring MVC**, **Spring Boot**, y otros módulos del ecosistema.

Principales Características de Spring Security

Autenticación y Autorización:

- ▶ Soporte para autenticación basada en formularios, HTTP Basic, OAuth2, JWT, etc.
- ▶ Gestión avanzada de roles y permisos.

Protección Automática Contra Vulnerabilidades:

- ▶ Prevención de ataques CSRF y XSS.
- ▶ Seguridad para cookies de sesión.

Principales Características de Spring Security

Integración con Bases de Datos y Servicios Externos:

- ▶ Gestión de usuarios y roles mediante JDBC, JPA, LDAP, OAuth2, etc.

Personalización:

- ▶ Configuraciones flexibles para adaptar la seguridad a las necesidades del proyecto.

Monitorización y Auditoría:

- ▶ Registro de eventos de autenticación y acceso.

Autenticación y Configuración Básica

Paso 1: Añadir Dependencias de Spring Security

- ▶ En el archivo pom.xml, añade la dependencia de Spring Security:

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

Autenticación y Configuración Básica

Paso 2: Configuración de la Base de Datos

- ▶ Crea una tabla en tu base de datos MySQL para almacenar usuarios y roles:

```
-- Tabla para usuarios
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    enabled BOOLEAN NOT NULL
);

-- Tabla para roles
CREATE TABLE roles (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    role VARCHAR(50) NOT NULL,
    FOREIGN KEY (username) REFERENCES users(username)
);
```

Autenticación y Configuración Básica

Paso 2: Configuración de la Base de Datos

- ▶ Inserta datos de prueba en las tablas:

```
-- Usuario 1: admin con rol ADMIN
INSERT INTO users (username, password, enabled)
VALUES ('admin', '$2a$10$VfJXZAvSOuASvEZYT3Hg5eHtYOsU.OytdHkBlt0T5rv5cA91tUx0S', true);

-- Usuario 2: user con rol USER
INSERT INTO users (username, password, enabled)
VALUES ('user', '$2a$10$7QQC1IX53w5MYd8W/kFhaOyzG7er84Wr.p0abZW3V0S6FGW0nEKCG', true);

-- Roles para cada usuario
INSERT INTO roles (username, role) VALUES ('admin', 'ROLE_ADMIN');
INSERT INTO roles (username, role) VALUES ('user', 'ROLE_USER');
```

- ▶ **Nota:** Las contraseñas están encriptadas usando BCrypt. Para encriptar contraseñas, puedes usar un generador en línea o código como se explica más adelante.

Autenticación y Configuración Básica

Paso 3: Configuración de Seguridad

- ▶ Crea una clase de configuración personalizada para Spring Security, que se conecte a la base de datos para autenticar a los usuarios.

```
@Configuration
public class SecurityConfig {

    private final DataSource dataSource;

    public SecurityConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Autenticación y Configuración Básica

Paso 3: Configuración de Seguridad

```
@Bean
public UserDetailsService userDetailsService() {
    JdbcUserDetailsManager userDetailsManager = new JdbcUserDetailsManager(dataSource);
    userDetailsManager.setUsersByUsernameQuery(
        "SELECT username, password, enabled FROM users WHERE username = ?");
    userDetailsManager.setAuthoritiesByUsernameQuery(
        "SELECT username, role FROM roles WHERE username = ?");
    return userDetailsManager;
}
```

- ▶ El método `userDetailsService` se utiliza para cargar los detalles de un usuario (como nombre, contraseña y roles) desde una fuente de datos personalizada, como una base de datos, para autenticación y autorización en Spring Security.

Autenticación y Configuración Básica

Paso 3: Configuración de Seguridad

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/public/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .httpBasic();
    return http.build();
}
```

- ▶ El método `filterChain` configura la cadena de filtros de seguridad en Spring Security, definiendo cómo se manejan las solicitudes HTTP, qué rutas requieren autenticación y qué mecanismos de autenticación (como formularios o HTTP Basic) se utilizarán.

Autenticación y Configuración Básica

Paso 4: Configurar application.properties

En el archivo src/main/resources/application.properties, configura la conexión a la base de datos (si no la tenemos ya configurada):

```
spring.datasource.url=jdbc:mysql://localhost:3306/spring_security_db
spring.datasource.username=tu_usuario
spring.datasource.password=tu_contraseña
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Autenticación y Configuración Básica

Paso 5: Crear un Controlador para Probar la Seguridad

- ▶ Crea un controlador REST con rutas públicas y protegidas:

```
@RestController
public class TestController {

    @GetMapping("/public/hello")
    public String publicEndpoint() {
        return "|Este es un endpoint público!";
    }

    @GetMapping("/private/hello")
    public String privateEndpoint() {
        return "|Este es un endpoint privado, necesitas autenticación!";
    }

    @GetMapping("/admin/hello")
    public String adminEndpoint() {
        return "|Este es un endpoint privado, solo accesible para administradores!";
    }
}
```

Autenticación y Configuración Básica

Paso 6: Probar la Aplicación

1. Acceso Público:

- ▶ Navega a `http://localhost:8080/public/hello` → No requiere autenticación.

2. Acceso Privado:

- ▶ Navega a `http://localhost:8080/private/hello` → Solicitará autenticación.
- ▶ Usa las credenciales:
 - ▶ Usuario: user | Contraseña: password.

3. Acceso Solo para Administradores:

- ▶ Navega a `http://localhost:8080/admin/hello`.
- ▶ Inicia sesión con:
 - ▶ Usuario: admin | Contraseña: password.

Autenticación y Configuración Básica

Encriptar Contraseñas con BCrypt

- ▶ Si necesitas encriptar contraseñas para almacenarlas en la base de datos, puedes usar este fragmento de código:

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class PasswordEncoderUtil {
    public static void main(String[] args) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        String rawPassword = "password";
        String encodedPassword = encoder.encode(rawPassword);
        System.out.println(encodedPassword);
    }
}
```

Autenticación y Configuración Básica

Ventajas de Esta Configuración

- ▶ Escalable: La autenticación basada en base de datos permite gestionar usuarios y roles dinámicamente.
- ▶ Segura: Contraseñas encriptadas con BCrypt.
- ▶ Flexible: Puedes combinar esta configuración con JWT, OAuth2 o cualquier otro método avanzado de autenticación.

Autenticación y Configuración con Anotaciones

Paso 1: Habilitar el Uso de Anotaciones

- ▶ Primero, habilita las anotaciones de seguridad en tu clase de configuración:

```
@Configuration  
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)  
public class SecurityConfig {  
    // Configuración general de seguridad (filterChain, userDetailsService, etc.)  
}
```

- ▶ `@EnableGlobalMethodSecurity`:
 - ▶ `prePostEnabled = true`: Habilita las anotaciones `@PreAuthorize` y `@PostAuthorize`.
 - ▶ `securedEnabled = true`: Habilita la anotación `@Secured`.
 - ▶ `jsr250Enabled = true` (opcional): Habilita la anotación `@RolesAllowed` (de JSR-250).

Autenticación y Configuración con Anotaciones

Paso 2: Usar Anotaciones en los Controladores

- ▶ Ahora puedes usar las anotaciones para proteger métodos en tus controladores:
 - ▶ **1. Con @PreAuthorize**
 - ▶ Esta anotación verifica condiciones antes de ejecutar el método. Por ejemplo:

```
@RestController
public class TestController {

    @GetMapping("/public")
    public String publicEndpoint() {
        return "Endpoint público, sin restricciones.";
    }

    @PreAuthorize("hasRole('USER')")
    @GetMapping("/private")
    public String privateEndpoint() {
        return "Endpoint privado, accesible solo para usuarios con rol USER.";
    }
}
```

Autenticación y Configuración con Anotaciones

Paso 2: Usar Anotaciones en los Controladores

- ▶ Ahora puedes usar las anotaciones para proteger métodos en tus controladores:
 - ▶ **2. Con @Secured**
 - ▶ Se utiliza para especificar roles que tienen acceso al método:

```
@RestController
public class TestController {

    @Secured("ROLE_USER")
    @GetMapping("/secured-user")
    public String securedForUser() {
        return "Acceso solo para usuarios con rol USER.";
    }
}
```

- ▶ **Nota:** Los roles deben estar prefijados con ROLE_ cuando se usan con @Secured.

Autenticación y Configuración con Anotaciones

Paso 2: Usar Anotaciones en los Controladores

► 3. Con @RolesAllowed

- Es una anotación estándar de JSR-250. Funciona similar a @Secured:

```
@RestController
public class TestController {

    @RolesAllowed("ROLE_USER")
    @GetMapping("/roles-user")
    public String rolesAllowedForUser() {
        return "Acceso solo para usuarios con rol USER.";
    }
}
```

Autenticación y Configuración con Anotaciones

Paso 2: Usar Anotaciones en los Controladores

- ▶ Puedes usar expresiones más complejas con `@PreAuthorize`:

```
@PreAuthorize("hasRole('ADMIN') or hasRole('USER')")
@GetMapping("/combined")
public String combinedAccess() {
    return "Acceso para usuarios con rol USER o ADMIN.";
}

@PreAuthorize("#username == authentication.name")
@GetMapping("/user/{username}")
public String personalAccess(@PathVariable String username) {
    return "Acceso permitido al usuario autenticado: " + username;
}
```

- ▶ Expresión `#username == authentication.name`:

- ▶ `authentication.name`: Es el nombre de usuario del usuario actualmente autenticado. Este valor es parte del objeto de autenticación que gestiona Spring Security.
- ▶ La condición verifica si el valor de `username` coincide con el nombre de usuario autenticado.

Conclusiones y Mejores Prácticas

Conclusiones y Mejores Prácticas

- ▶ **Separación de Responsabilidades**
 - ▶ El patrón MVC facilita la organización del código, mejorando la mantenibilidad y prueba.
- ▶ **Simplicidad con Spring Boot**
 - ▶ Configuración automática y herramientas como DevTools aceleran el desarrollo.
- ▶ **Desarrollo RESTful**
 - ▶ Diseña APIs claras y bien estructuradas siguiendo principios REST.
- ▶ **Gestión de Datos**
 - ▶ Spring Data simplifica las operaciones con bases de datos mediante repositorios y consultas personalizadas.
- ▶ **Seguridad Integrada**
 - ▶ Spring Security protege endpoints críticos y garantiza autenticación robusta con facilidad.

Conclusiones y Mejores Prácticas

- ▶ **Pruebas y Estabilidad**
 - ▶ Implementa pruebas unitarias e integradas para garantizar la calidad del código.
- ▶ **Despliegue Escalable**
 - ▶ Diseña pensando en entornos cloud-native y adopta contenedores como Docker.
- ▶ **Código Limpio y Mantenible**
 - ▶ Sigue principios SOLID, organiza las capas del proyecto y usa configuraciones centralizadas.
- ▶ **Manejo de Errores**
 - ▶ Usa controladores de excepciones globales (@ControllerAdvice) para manejar errores de manera uniforme en toda la aplicación. Proporciona mensajes de error claros y específicos para mejorar la experiencia del usuario

Conclusiones y Mejores Prácticas

- ▶ **Seguridad Avanzada:**
 - ▶ Mantén tus dependencias actualizadas para prevenir vulnerabilidades conocidas.
 - ▶ Adopta HTTPS en entornos de producción y protege los datos sensibles.
- ▶ **Optimización de Vistas:**
 - ▶ Usa motores de plantillas como Thymeleaf con fragmentos reutilizables para construir interfaces consistentes.
 - ▶ Adopta estrategias de carga diferida o paginación para manejar grandes volúmenes de datos.

Conclusiones y Mejores Prácticas

- ▶ **Spring MVC** es una herramienta integral que, combinada con **Spring Boot**, permite desarrollar aplicaciones modernas y escalables de forma eficiente.
- ▶ Siguiendo buenas prácticas, aplicando seguridad y organizando el proyecto correctamente, es posible construir aplicaciones web robustas que respondan a las necesidades actuales de los desarrolladores y usuarios.

¿Alguna duda?



Muchas gracias 😊

Antonio Varela Nieto