Spring Framework

Índice de contenidos

Integrando Spring con bases de datos: H2, MySQL, NoSQL
    - Integrando Spring con Sesiones
    - Integrando Spring con servicios web RESTful
- Programación orientada a aspectos (AOP)
    - Entendiendo la AOP
    - Implementando AOP usando Spring
    - POA con Spring Security
- Pruebas y despliegue
    - Pruebas de aplicaciones Spring
    - Despliegue de aplicaciones Spring
    - Optimización de rendimiento de aplicaciones Spring
- Spring Security
    - Visión general de Spring Security
    - Autenticación y autorización
    - Seguridad de aplicaciones web
- Spring Batch
    - Visión general de Spring Batch
    - Arquitectura de Spring Batch
    - Spring Batch con Spring MVC
- Spring Web Flow
    - Visión general de Spring Web Flow
    - Arquitectura de Spring Web Flow
    - Spring Web Flow con Spring MVC

Introducción al Framework Spring

Spring Framework es un framework de código abierto para el desarrollo de aplicaciones empresariales en Java. Fue creado por Rod Johnson en 2002.

El framework se basa en la inversión de control (IoC) y la inyección de dependencias (DI), lo que permite una separación clara entre la lógica de la aplicación y la infraestructura subyacente. Esto permite a los desarrolladores centrarse en la lógica de negocio de la aplicación, sin tener que preocuparse por la infraestructura subyacente.

Spring Boot es un framework de Java que se basa en Spring Framework y se utiliza para desarrollar aplicaciones web y móviles de manera rápida y sencilla. Fue creado en 203 por Pivotal Software y se ha

convertido en una de las herramientas más populares para el desarrollo de aplicaciones empresariales en Java.

Visión general del Framework Spring

Spring Boot es un framework de Java diseñado para facilitar el desarrollo de aplicaciones empresariales de manera rápida y sencilla. Este framework se basa en Spring Framework, pero se diferencia en que utiliza una estrategia de "opiniónated" para la configuración y un enfoque de "convención sobre configuración" que reduce la cantidad de código que los desarrolladores necesitan escribir.

Una de las principales características de Spring Boot es la configuración automática, que simplifica la configuración de la aplicación al eliminar la necesidad de configuración manual. Spring Boot utiliza anotaciones y metadatos para configurar automáticamente la aplicación, lo que permite a los desarrolladores centrarse en la lógica de la aplicación en lugar de la configuración.

Además, Spring Boot es altamente modular y se integra fácilmente con otros frameworks y tecnologías. Esto significa que los desarrolladores pueden utilizar diferentes tecnologías según sus necesidades sin tener que preocuparse por la compatibilidad y la configuración.

Otra característica importante de Spring Boot es su capacidad para crear aplicaciones autocontenidas y ejecutables en un solo archivo JAR. Esto significa que las aplicaciones se pueden desplegar fácilmente en diferentes entornos sin tener que preocuparse por las dependencias y configuraciones externas.

En resumen, Spring Boot es un framework eficiente que simplifica el desarrollo de aplicaciones empresariales en Java.

Arquitectura de Spring Boot

La arquitectura de Spring Boot se basa en el framework Spring, que sigue el patrón de diseño de inversión de control (IoC) y el principio de “convención sobre configuración”. Spring Boot utiliza anotaciones para definir las capas de la aplicación, como Repository, Service y Component. Estas anotaciones permiten a Spring Boot crear automáticamente los objetos necesarios y establecer las dependencias entre ellos. Además, Spring Boot ofrece una serie de herramientas para facilitar el desarrollo, como los starters, que son dependencias predefinidas para diferentes tipos de proyectos, o el autoconfigurador, que adapta la configuración según las dependencias disponibles. Con Spring Boot se puede crear una aplicación autoejecutable que contiene un servidor web embebido.

Módulos de Spring Boot

Spring Framework se compone de varios módulos que ofrecen diferentes servicios y funcionalidades. Algunos de los paquetes más importantes de Spring Framework son:

- Spring Core: proporciona soporte para la inversión de control (IoC) y la inyección de dependencias (DI).
- Spring JDBC: proporciona soporte para la conexión con bases de datos relacionales.
- Spring ORM: proporciona soporte para la conexión con bases de datos relacionales y NoSQL.
- Spring Web: proporciona soporte para la creación de aplicaciones web.
- Spring MVC: proporciona soporte para la creación de aplicaciones web basadas en el patrón de diseño MVC.

Spring AOP: proporciona soporte para la programación orientada a aspectos (AOP). *Spring Session: proporciona soporte para la gestión de sesiones.

- Spring Security: proporciona soporte para la seguridad de aplicaciones web.
- Spring Batch: proporciona soporte para la creación de aplicaciones de procesamiento de lotes.
- Spring Web Flow: proporciona soporte para la creación de aplicaciones web basadas en el patrón de diseño de flujo de trabajo.
- Spring Integration: proporciona soporte para la integración con otras tecnologías y frameworks.
- Spring Data: proporciona soporte para la conexión con bases de datos relacionales y NoSQL.
- Spring Test: proporciona soporte para la creación de pruebas unitarias y de integración.
- Spring Boot: proporciona soporte para la creación de aplicaciones autocontenidas y ejecutables en un solo archivo JAR.
- Spring Cloud: proporciona soporte para la creación de aplicaciones basadas en microservicios.

Ventajas del Framework Spring Boot

Ventajas de Spring Boot frente a Spring Framework

- **Configuración automática**: Spring Boot utiliza una estrategia de configuración automática, lo que significa que configura la aplicación de manera predeterminada y elimina la necesidad de escribir código de configuración manual.
- **Menos código**: Spring Boot permite a los desarrolladores crear aplicaciones con menos código en comparación con Spring Framework. Esto se debe a que Spring Boot viene con configuraciones predefinidas y dependencias incorporadas.
- **Integración simplificada**: Spring Boot simplifica la integración con otras tecnologías y frameworks.
- **Enfoque de "opinionated"**: lo que significa que proporciona soluciones predefinidas para la mayoría de los casos de uso comunes.
- **Microservicios**: Spring Boot es ideal para el desarrollo de aplicaciones de microservicios, lo que significa que los desarrolladores pueden crear aplicaciones modulares y escalables que sean fáciles de mantener y actualizar.
- **Despliegue sencillo**: Spring Boot permite a los desarrolladores crear aplicaciones autocontenidas y ejecutables en un solo archivo JAR. Esto significa que las aplicaciones se pueden desplegar fácilmente en diferentes entornos sin tener que preocuparse por las dependencias y configuraciones externas.

Creación de aplicaciones Spring Boot

Spring Boot es un framework de Java que se utiliza para crear aplicaciones autocontenidas y ejecutables en un solo archivo JAR. Este framework se basa en Spring Framework, pero se diferencia en que utiliza una estrategia de "opiniónated" para la configuración y un enfoque de "convención sobre configuración" que reduce la cantidad de código que los desarrolladores necesitan escribir.

Creación de una aplicación Spring Boot

Para crear una aplicación Spring Boot, hay que seguir los siguientes pasos:

- Instalar Spring Boot: puedes usar Spring Boot Starter, que es un Eclipse con Spring Boot integrado, o instalar Spring Tools en tu IDE preferido.
- Crear un proyecto: puedes usar el asistente de Spring Boot para generar un proyecto con las dependencias y la configuración necesarias, o usar el sitio web https://start.spring.io/ para descargar un proyecto inicializado.
- Desarrollar los servicios: puedes crear clases Java con anotaciones de Spring para definir los controladores, los servicios y los repositorios que componen tu aplicación. También puedes usar otras tecnologías como Thymeleaf o JPA para crear vistas o acceder a bases de datos.
- Probar la aplicación: puedes ejecutar la aplicación desde tu IDE o desde la línea de comandos usando el comando `mvn spring-boot:run` o `java -jar nombre-del-archivo.jar`. Luego puedes acceder a la aplicación desde tu navegador usando la dirección http://localhost:8080/.

Creación de una aplicación Spring Boot con Spring Initializr

Para crear una aplicación Spring Boot con Spring Initializr, puedes seguir estos pasos:

- Accede al sitio web https://start.spring.io/.
- Elige si quieres usar Maven o Gradle para la construcción de tu proyecto.
- Especifica la versión de Spring Boot y de Java que quieres usar.
- Introduce los datos de tu proyecto, como el grupo, el artefacto, el nombre y la descripción.
- Selecciona las dependencias que necesitas para tu aplicación. Puedes buscarlas por nombre o por categoría.
- Haz clic en Generate para descargar el proyecto en un archivo ZIP.
- Extrae los archivos del ZIP en una carpeta local y abre el proyecto con tu IDE preferido.

Spring Boot con Maven

Para gestionar las tareas de compilación, ejecución, limpieza y testing en Spring Boot con Maven, puedes usar los siguientes comandos:

- Para compilar tu aplicación, usa el comando `mvn compile`.
- Para ejecutar tu aplicación, usa el comando `mvn spring-boot:run`. Asegúrate de estar en la carpeta del archivo JAR que se encuentra en la carpeta "target".
- Para limpiar tu proyecto, usa el comando `mvn clean`. Esto eliminará los archivos generados por la compilación.
- Para realizar los tests de tu aplicación, usa el comando `mvn test`. Puedes usar las anotaciones y funcionalidades que te proporciona Spring Boot para crear tests unitarios e integrados.

-El archivo POM de tu proyecto debe tener las siguientes dependencias:

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

Spring Boot con Gradle

Para gestionar las tareas de compilación, ejecución, limpieza y testing en Spring Boot con Gradle, puedes usar los siguientes comandos:

- Para compilar tu aplicación, usa el comando `gradle build`. Este comando descargará las dependencias, construirá las clases, ejecutará los tests y empaquetará las clases en un archivo JAR.
- Para ejecutar tu aplicación, usa el comando `gradle bootRun`. Este comando ejecutará tu aplicación en forma explosionada.
- Para limpiar tu proyecto, usa el comando `gradle clean`. Este comando eliminará los archivos generados por la compilación.
- Para realizar los tests de tu aplicación, usa el comando `gradle test`. Puedes usar las anotaciones y funcionalidades que te proporciona Spring Boot para crear tests unitarios e integrados.

El archivo build.gradle de tu proyecto debe tener las siguientes dependencias:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
}
```

Spring Core

Visión general de Spring Core

Spring Core es el módulo central del framework Spring que proporciona las funcionalidades básicas para el desarrollo de aplicaciones Java empresariales. Spring Core se basa en el concepto de **inyección de dependencias**, que es un patrón de diseño que permite desacoplar los componentes de una aplicación y facilitar su configuración, reutilización y testing. Spring Core también ofrece otras características como el soporte para anotaciones, la gestión del ciclo de vida de los objetos, la integración con otras tecnologías y el acceso a recursos externos.

Spring Core es el motor del framework Spring y el fundamento sobre el que se construyen los demás módulos. Conocer Spring Core te permitirá aprovechar todo el potencial de esta plataforma para crear aplicaciones profesionales, escalables y mantenibles.

Inyección de dependencias

La inyección de dependencias en Spring Core es un patrón de diseño que permite que los objetos no sean responsables de crear e inicializar sus dependencias, sino que estas sean provistas por otro objeto. En el caso de Spring, ese objeto es el **contenedor IoC**, que es el encargado de gestionar el ciclo de vida y la configuración de los objetos que forman parte de la aplicación.

La inyección de dependencias en Spring Core se puede realizar mediante diferentes modos, como el uso de anotaciones, archivos XML o clases Java. El modo más común y recomendado es mediante anotaciones, que son etiquetas que se colocan sobre las clases o los atributos para indicar al contenedor IoC cómo debe crear e inyectar las dependencias.

Inversión de control

La inversión de control (IoC) es un patrón de diseño que consiste en delegar el control de la creación, inicialización y conexión de los objetos de una aplicación a un objeto externo llamado **contenedor**. El contenedor se encarga de gestionar el ciclo de vida y la configuración de los objetos, que son llamados **beans**, y proveerlos cuando sean necesarios a través de la **inyección de dependencias**.

En Spring Boot, el contenedor IoC está representado por la interfaz **ApplicationContext**, que se puede configurar mediante anotaciones, archivos XML o clases Java. El contenedor IoC lee la configuración y crea e inyecta los beans según las anotaciones o las definiciones que se hayan especificado.

La inversión de control en Spring Boot permite desacoplar los componentes de una aplicación y facilitar su reutilización, testing y mantenimiento. Además, permite aprovechar las características adicionales que ofrece el framework Spring a través del tiempo de vida de los objetos.

Principales anotaciones de Spring Core

Las principales anotaciones de Spring Core son las siguientes:

- `@SpringBootApplication`: indica que una clase es la principal de una aplicación spring boot y habilita la configuración automática, el escaneo de componentes y otras características.
- `@Configuration`: indica que una clase define beans adicionales o importa otras clases de configuración para el contexto de spring.
- `@EnableAutoConfiguration`: indica que se debe usar el mecanismo de configuración automática de spring boot para inferir los beans necesarios según las dependencias del classpath.
- `@ComponentScan`: indica el paquete o los paquetes donde buscar beans anotados con @Component, @Repository, @Service o @Controller.
- `@Component`: indica que una clase es un componente gestionado por el contenedor IoC y puede ser inyectado como una dependencia.
- `@Autowired`: indica que un atributo debe ser inyectado con una instancia del tipo adecuado creada por el contenedor IoC.
- `@Qualifier`: indica el nombre específico del componente que se quiere inyectar cuando hay más de uno del mismo tipo.
- `@Value`: indica que un atributo debe ser inyectado con un valor literal o una expresión.
- `@Required`: se aplica a un método setter y indica que el bean anotado debe llenarse en el momento de la configuración con la propiedad requerida, de lo contrario arroja una excepción BeanInitilizationException.

Además, existen otras anotaciones que son especializaciones de `@Component` y que categorizan los componentes según su responsabilidad. Estas son:

- `@Controller`: indica que una clase es un controlador web y puede manejar peticiones HTTP.
- `@Service`: indica que una clase es un servicio de negocio y contiene la lógica de aplicación.
- `@Repository`: indica que una clase es un repositorio de datos y puede acceder a la capa de persistencia.

Un ejemplo de una clase que se inyecta en otra clase mediante anotaciones:

```
//Clase principal de la aplicación
@SpringBootApplication //Indica que es la clase principal y habilita la configuración
automática y el escaneo de componentes
public class Aplicacion {

    public static void main(String[] args) {
        SpringApplication.run(Aplicacion.class, args); //Inicia la aplicación spring boot
    }
}

//Clase de configuración
@Configuration //Indica que define beans adicionales o importa otras clases de configuración
public class Configuracion {

    @Bean //Indica que el método devuelve un bean que debe ser registrado en el contexto de
spring
    public Servicio servicio() {
        return new Servicio(); //Crea una instancia del servicio
    }
}

//Clase de servicio
@Service //Indica que es un bean de la capa de negocio
public class Servicio {

    public String saludo() {
        return "Hola desde el servicio"; //Devuelve un saludo
    }
}

//Clase de controlador
@Controller //Indica que es un controlador para spring MVC
public class Controlador {

    @Autowired //Indica que la propiedad debe ser inyectada con otro bean de spring (en este
caso el servicio)
    private Servicio servicio;

    @GetMapping("/") //Indica que el método maneja las peticiones GET a la ruta "/"
    public String inicio(Model model) {
        model.addAttribute("saludo", servicio.saludo()); //Añade el saludo del servicio al
modelo
        return "inicio"; //Devuelve el nombre de la vista a mostrar (en este caso
"inicio.html")
    }
}
```

Spring Web

Visión general de Spring Web

De acuerdo, puedo darte una visión general del paquete Spring Web en el framework Spring Boot. Según los resultados de búsqueda[1][2][3], el paquete Spring Web es uno de los starters que ofrece Spring Boot para facilitar el desarrollo de aplicaciones web con Spring. Al incluir este paquete en el proyecto, se obtienen las siguientes ventajas:

- Se configura automáticamente un servidor web embebido (por defecto Tomcat) que se puede arrancar con un solo comando.
- Se habilita el soporte para las anotaciones de Spring MVC y se registran automáticamente los controladores y otros componentes web.
- Se proporciona una integración con otras tecnologías web como Thymeleaf, JSP o JSF.
- Se simplifica la seguridad web con la integración de Spring Security y Azure Active Directory.

El paquete Spring Web es una forma rápida y sencilla de crear aplicaciones web con Spring Boot sin tener que configurar manualmente muchos aspectos. Puedes encontrar más información sobre el paquete Spring Web en la documentación oficial[4].

Anotaciones de Spring Web

Las principales anotaciones de Spring Web son las siguientes:

- **@RequestMapping**: Esta anotación marca los métodos controladores dentro de las clases @Controller y se puede configurar usando path, name o value para indicar a qué URL se asigna el método.
- **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping**: Estas anotaciones son alias de @RequestMapping con diferentes métodos HTTP y se utilizan para asignar métodos controladores a diferentes rutas URL según el tipo de solicitud.
- **@PathVariable**: Esta anotación se utiliza para vincular una parte de la URL a un parámetro del método controlador.
- **@RequestParam**: Esta anotación se utiliza para vincular un parámetro de consulta o un parámetro de formulario a un parámetro del método controlador.
- **@RequestBody**: Esta anotación se utiliza para vincular el cuerpo de la solicitud HTTP a un objeto en el método controlador.
- **@ResponseBody**: Si marcamos un método controlador con @ResponseBody, Spring trata el resultado del método como la respuesta misma y lo convierte al formato adecuado según el tipo de contenido de la solicitud.
- **@ExceptionHandler**: Con esta anotación, podemos declarar un método personalizado para manejar los errores que se produzcan en los métodos controladores y devolver una respuesta adecuada al cliente.
- **@ResponseStatus**: Podemos especificar el estado HTTP deseado de la respuesta si anotamos un método controlador con esta anotación. También podemos usarla junto con @ExceptionHandler para

indicar el estado HTTP del error.

Ejemplo de aplicación web con Spring Boot

Ejemplo de aplicación web con Spring Boot:

```
@SpringBootApplication
public class Aplicacion {

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

@Controller
public class Controlador {

    @GetMapping("/")
    public String inicio(Model model) {
        model.addAttribute("saludo", "Hola desde el controlador");
        return "inicio";
    }
}
```

Un ejemplo del uso de la anotación @RequestMapping:

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

    // GET /users -> devuelve una lista de usuarios
    @RequestMapping(method = RequestMethod.GET)
    public String getUsers(Model model) {
        List<User> users = userService.getUsers();
        model.addAttribute("users", users);
        return "users";
    }

    // POST /users -> crea un nuevo usuario
    @RequestMapping(method = RequestMethod.POST)
    public String createUser(@ModelAttribute User user) {
        userService.createUser(user);
        return "redirect:/users";
    }

    // PUT /users/{id} -> actualiza un usuario existente
    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    public String updateUser(@PathVariable Long id, @ModelAttribute User user) {
        userService.updateUser(id, user);
        return "redirect:/users";
    }

    // DELETE /users/{id} -> elimina un usuario existente
    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public String deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return "redirect:/users";
    }
}
```

Un ejemplo del uso de las anotaciones @GetMapping, @PostMapping, @PutMapping y @DeleteMapping:

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

    private List<User> users = new ArrayList<>();

    //Obtener todos los usuarios
    @GetMapping("/")
    public List<User> getUsers() {
        return users;
    }

    //Obtener un usuario por su ID
    @GetMapping("/{id}")
    public User getUserById(@PathVariable int id) {
        return users.stream()
                .filter(user -> user.getId() == id)
                .findFirst()
                .orElse(null);
    }

    //Agregar un nuevo usuario
    @PostMapping("/")
    public User addUser(@RequestBody User user) {
        users.add(user);
        return user;
    }

    //Actualizar un usuario existente
    @PutMapping("/{id}")
    public User updateUser(@PathVariable int id, @RequestBody User updatedUser) {
        User user = getUserById(id);
        if (user != null) {
            user.setName(updatedUser.getName());
            user.setEmail(updatedUser.getEmail());
        }
        return user;
    }

    //Eliminar un usuario
    @DeleteMapping("/{id}")
    public User deleteUser(@PathVariable int id) {
        User user = getUserById(id);
        if (user != null) {
            users.remove(user);
        }
        return user;
    }

    //Actualizar parcialmente un usuario
    @PatchMapping("/{id}")
    public User partialUpdateUser(@PathVariable int id, @RequestBody Map<String, Object>
updates) {
        User user = getUserById(id);
        if (user != null) {
            updates.forEach((key, value) -> {
                switch (key) {
                    case "name":
                        user.setName((String) value);
```

Spring Data

Visión general de Spring Data

Spring Data es un subproyecto de Spring Framework que proporciona un conjunto de bibliotecas y herramientas para simplificar el acceso a los datos de diferentes tipos de bases de datos, como bases de datos relacionales y no relacionales.

El objetivo principal de Spring Data es abstraer la complejidad del acceso a los datos y proporcionar una API coherente y fácil de usar para interactuar con diferentes bases de datos. Esto significa que los desarrolladores pueden concentrarse en la lógica de negocio en lugar de preocuparse por cómo acceder a los datos.

Spring Data proporciona una amplia gama de funcionalidades, desde la creación de repositorios que permiten realizar operaciones CRUD (crear, leer, actualizar y eliminar) de forma rápida y fácil, hasta la implementación de consultas complejas con lenguajes específicos de la base de datos.

Además, Spring Data es altamente personalizable y extensible, lo que significa que puedes adaptarlo a tus necesidades específicas. Por ejemplo, puedes agregar tus propios métodos de consulta personalizados o usar tus propias anotaciones para mapear tus entidades de dominio a la base de datos.

En resumen, Spring Data es una herramienta muy útil para simplificar el acceso a los datos en aplicaciones de Spring. Al utilizar Spring Data, puedes escribir menos código, reducir la complejidad y aumentar la productividad.

Anotaciones de Spring Data

Las anotaciones de Spring Data:

- @Repository: Anota una clase que proporciona el mecanismo de persistencia para trabajar con una base de datos. Esta anotación es opcional, pero se recomienda su uso para facilitar la inyección de dependencias y la transparencia transaccional.
- @Table: Anota una clase que representa una tabla en una base de datos relacional. Esta anotación es utilizada por el ORM (Object-Relational Mapping) para mapear los objetos Java a las tablas de la base de datos.
- @Entity: Anota una clase que representa una tabla en una base de datos relacional. Esta anotación es utilizada por el ORM (Object-Relational Mapping) para mapear los objetos Java a las tablas de la base de datos.
- @Id: Anota el atributo que se utilizará como clave primaria en una tabla de base de datos. Esta anotación es necesaria para que el ORM pueda identificar los registros de la tabla.
- @GeneratedValue: Anota el atributo que será generado automáticamente por la base de datos cuando se inserte un nuevo registro. Esta anotación se utiliza en combinación con @Id.
- @Column: Anota un atributo que representa una columna en una tabla de base de datos. Esta anotación es utilizada por el ORM para mapear los atributos de la clase a las columnas de la tabla.
- @Transient: Anota un atributo que no se debe mapear a una columna en la tabla de base de datos. Esta anotación se utiliza en combinación con @Column.
-

@Transactional: Anota un método o una clase que ejecuta operaciones transaccionales. Esta anotación asegura que todas las operaciones en el método o la clase se ejecuten en una única transacción.

- @Query: Anota un método que ejecuta una consulta personalizada. Esta anotación se utiliza en combinación con @Modifying para ejecutar consultas de actualización.
- @Modifying: Anota un método que ejecuta una consulta de actualización. Esta anotación se utiliza en combinación con @Query para ejecutar consultas personalizadas.
- @Param: Anota un parámetro de un método que ejecuta una consulta personalizada. Esta anotación se utiliza en combinación con @Query para ejecutar consultas personalizadas.

Ejemplo de aplicación con Spring Data

Ejemplo de aplicación con Spring Data:

```
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

    @Column(name = "description")
    private String descripcion;

    @Column(name = "price")
    private Double precio;

    @CreatedDate
    private LocalDateTime fechaCreacion;

    @LastModifiedDate
    private LocalDateTime fechaModificacion;

    // constructor, getters y setters
}

@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {
    @Query("SELECT p FROM Producto p WHERE p.precio > :precio")
    List<Producto> buscarPorPrecioMayorQue(@Param("precio") Double precio);

    @Modifying
    @Query("UPDATE Producto p SET p.precio = :precio WHERE p.id = :id")
    void actualizarPrecioPorId(@Param("id") Long id, @Param("precio") Double precio);
}

@Service
public class ProductoService {
    @Autowired
    private ProductoRepository productoRepository;

    @Transactional
    public void actualizarPrecioPorId(Long id, Double precio) {
        productoRepository.actualizarPrecioPorId(id, precio);
    }

    public List<Producto> buscarPorPrecioMayorQue(Double precio) {
        return productoRepository.buscarPorPrecioMayorQue(precio);
    }
}
```

Spring MVC

Visión general de Spring MVC

Spring MVC utiliza el patrón de diseño Modelo-Vista-Controlador para separar la lógica de la aplicación en tres componentes: el modelo, la vista y el controlador. El modelo representa los datos y la lógica de la aplicación, la vista es la interfaz de usuario y el controlador maneja las solicitudes HTTP y coordina la interacción entre el modelo y la vista.

En Spring MVC, el controlador es responsable de manejar las solicitudes HTTP y de devolver la vista adecuada. El modelo se utiliza para almacenar los datos de la aplicación y la vista se utiliza para mostrar los datos al usuario.

Spring MVC es muy popular en el mundo Java y se utiliza en muchas aplicaciones web. Es fácil de usar y ofrece muchas características útiles, como la validación de formularios y la internacionalización.

Anotaciones de Spring MVC

Las anotaciones de Spring MVC:

- **@Controller**: Se utiliza para marcar una clase como un controlador en Spring MVC. Los métodos en un controlador marcados con esta anotación manejan las solicitudes HTTP entrantes.
- **@RequestMapping**: Se utiliza para mapear una solicitud HTTP a un método en un controlador. La anotación especifica la URL de la solicitud y el método HTTP que debe manejarla.
- **@PathVariable**: Se utiliza para mapear una parte de una URL a un parámetro de método en un controlador.
- **@RequestParam**: Se utiliza para mapear un parámetro de solicitud HTTP a un parámetro de método en un controlador.
- **@ModelAttribute**: Se utiliza para mapear un objeto Java a un modelo que se utiliza en una vista de Spring MVC.
- **@ResponseBody**: Se utiliza para indicar que un método en un controlador debe devolver el cuerpo de la respuesta HTTP.
- **@ResponseStatus**: Se utiliza para especificar el código de estado HTTP que debe devolver un método en un controlador.
- **@SessionAttribute**: Se utiliza para marcar un atributo como un atributo de sesión en Spring MVC.
- **@InitBinder**: Se utiliza para personalizar el enlace de datos en Spring MVC.
- **@ExceptionHandler**: Se utiliza para manejar excepciones específicas en Spring MVC.

Ejemplo de aplicación con Spring MVC

Tenemos una clase que modela una tarea:

```
public class Tarea {
    private Long id;
    private String titulo;
    private String descripcion;
    private Boolean completada;

    // constructor, getters y setters
}
```

Tenemos una interfaz que extiende de JpaRepository para acceder a los datos de la base de datos:

```
public interface TareaRepository extends JpaRepository<Tarea, Long> {
}
```

Tenemos una clase de servicio que utiliza la interfaz de repositorio para acceder a los datos de la base de datos:

```
@Service
public class TareaService {
    @Autowired
    private TareaRepository tareaRepository;

    public List<Tarea> buscarTodas() {
        return tareaRepository.findAll();
    }

    public Tarea buscarPorId(Long id) {
        return tareaRepository.findById(id).orElse(null);
    }

    public void guardar(Tarea tarea) {
        tareaRepository.save(tarea);
    }

    public void eliminar(Long id) {
        tareaRepository.deleteById(id);
    }
}
```

Tenemos una clase de controlador que utiliza la clase de servicio para acceder a los datos de la base de datos:

```
@Controller

public class TareaController {
    @Autowired
    private TareaService tareaService;

    @GetMapping("/")
    public String mostrarTodas(Model model) {
        model.addAttribute("tareas", tareaService.buscarTodas());
        return "index";
    }

    @GetMapping("/nueva")
    public String mostrarFormularioNuevaTarea(Model model) {
        Tarea tarea = new Tarea();
        model.addAttribute("tarea", tarea);
        return "nueva_tarea";
    }

    @PostMapping("/guardar")
    public String guardarTarea(@ModelAttribute("tarea") Tarea tarea) {
        tareaService.guardar(tarea);
        return "redirect:/";
    }

    @GetMapping("/editar/{id}")
    public String mostrarFormularioEditarTarea(@PathVariable(value = "id") Long id, Model
model) {
        Tarea tarea = tareaService.buscarPorId(id);
        model.addAttribute("tarea", tarea);
        return "editar_tarea";
    }

    @GetMapping("/eliminar/{id}")
    public String eliminarTarea(@PathVariable(value = "id") Long id) {
        tareaService.eliminar(id);
        return "redirect:/";
    }
}
```

nueva_tarea.html:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Nueva tarea</title>
</head>

<body>
    <h1>Nueva tarea</h1>
    <form action="#" th:action="@{/guardar}" th:object="${tarea}" method="post">
        <label for="titulo">Título</label>
        <input type="text" id="titulo" th:field="*{titulo}"/>
        <br/>
        <label for="descripcion">Descripción</label>
        <input type="text" id="descripcion" th:field="*{descripcion}"/>
        <br/>
        <label for="completada">Completada</label>
        <input type="checkbox" id="completada" th:field="*{completada}"/>
        <br/>
        <button type="submit">Guardar</button>
    </form>
</body>
</html>
```

editar_tarea.html:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Editar tarea</title>
</head>

<body>
    <h1>Editar tarea</h1>
    <form action="#" th:action="@{/guardar}" th:object="${tarea}" method="post">
        <label for="titulo">Título</label>
        <input type="text" id="titulo" th:field="*{titulo}"/>
        <br/>
        <label for="descripcion">Descripción</label>
        <input type="text" id="descripcion" th:field="*{descripcion}"/>
        <br/>
        <label for="completada">Completada</label>
        <input type="checkbox" id="completada" th:field="*{completada}"/>
        <br/>
        <button type="submit">Guardar</button>
    </form>
</body>
</html>
```

index.html:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Lista de tareas</title>
</head>

<body>
    <h1>Lista de tareas</h1>
    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Título</th>
                <th>Descripción</th>
                <th>Completada</th>
                <th>Acciones</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="tarea : ${tareas}">
                <td th:text="${tarea.id}"></td>
                <td th:text="${tarea.titulo}"></td>
                <td th:text="${tarea.descripcion}"></td>
                <td th:text="${tarea.completada}"></td>
                <td>
                    <a th:href="@{/editar/{id}(id=${tarea.id})}">Editar</a>
                    <a th:href="@{/eliminar/{id}(id=${tarea.id})}">Eliminar</a>
                </td>
            </tr>
        </tbody>
    </table>
    <a th:href="@{/nueva}">Nueva tarea</a>
</body>
</html>
```