Spring Framework

Índice de contenidos

    - 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.

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);
                        break;
                    case "email":
                        user.setEmail((String) value);
                        break;
                }
            });
        }
        return user;
    }
}
```

Validación de datos

La validación de datos es un proceso que se utiliza para verificar que los datos de entrada sean correctos y cumplan con los requisitos establecidos. Por ejemplo, si tenemos un formulario de registro de usuarios, podemos usar la validación de datos para verificar que el nombre de usuario no esté vacío, que la contraseña tenga al menos 8 caracteres, que el correo electrónico tenga un formato válido, etc.

Spring Boot proporciona una serie de anotaciones que podemos usar para validar los datos de entrada en los métodos controladores. Estas anotaciones se pueden aplicar a los parámetros de los métodos controladores o a los campos de los objetos de modelo.

Las anotaciones de validación de datos numéricos son:

- **@NotNull**: El campo no puede ser nulo.
- **@NotEmpty**: El campo no puede ser nulo ni vacío.
- **@NotBlank**: El campo no puede ser nulo ni estar en blanco.
- **@Size**: El campo debe tener un tamaño entre los valores especificados.
- **@Min**: El campo debe tener un valor mayor o igual al especificado.
- **@Max**: El campo debe tener un valor menor o igual al especificado.
- **@Valid**: El campo debe ser válido.
- **@DecimalMax**: El campo debe tener un valor menor o igual al especificado.
- **@DecimalMin**: El campo debe tener un valor mayor o igual al especificado.
- **@Digits**: El campo debe tener un número de dígitos (enteros y decimales) menor o igual al especificado.
- **@Negative**: El campo debe tener un valor negativo.
- **@NegativeOrZero**: El campo debe tener un valor negativo o cero.
- **@Positive**: El campo debe tener un valor positivo.
- **@PositiveOrZero**: El campo debe tener un valor positivo o cero.

Las anotaciones de validación de datos de cadena son:

- **@Email**: El campo debe tener un formato de correo electrónico válido.
- **@Pattern**: El campo debe coincidir con el patrón especificado.
- **@URL**: El campo debe tener un formato de URL válido.
- **@NotBlank**: El campo no puede ser nulo ni estar en blanco.
- **@NotEmpty**: El campo no puede ser nulo ni vacío.
- **@Size**: El campo debe tener un tamaño entre los valores especificados.

Las anotaciones de validación de datos de fecha son:

- **@Future**: El campo debe ser una fecha futura.
- **@FutureOrPresent**: El campo debe ser una fecha futura o la fecha actual.

- **@Past**: El campo debe ser una fecha pasada.
- **@PastOrPresent**: El campo debe ser una fecha pasada o la fecha actual.

Las anotaciones de validación de datos de tipo booleano son:

- **@AssertTrue**: El campo debe ser verdadero.
- **@AssertFalse**: El campo debe ser falso.

Un ejemplo de validación de datos con anotaciones:

```
@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 @Valid User user) {
        users.add(user);
        return user;
    }

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

```
                        break;
                }
            });
        }
        return user;
    }
}
```

Donde User.java es:

```
public class User {

    private int id;
    @NotBlank
    private String name;
    @Email
    private String email;

    //Constructores, getters y setters
}
```

Un ejemplo de Entity con muchas de las anotaciones de validación de datos que se pueden usar:

```
@Entity
@Table(name = "users")
public class User {

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

    @NotBlank
    @Size(min = 3, max = 50)
    private String name;

    @NotBlank
    @Size(min = 3, max = 50)
    @Email
    private String email;

    @NotBlank
    @Size(min = 8, max = 100)
    private String password;

    @NotNull
    @Min(18)
    @Max(100)
    private int age;

    @NotNull
```

```
    @PastOrPresent
    private LocalDate birthDate;

    @NotNull
    @AssertTrue
    private boolean terms;

    //Constructores, getters y setters
}
```

Manejo de errores

Cuando se produce un error en la aplicación, Spring Boot devuelve una respuesta HTTP con un código de estado de error y un mensaje de error. Por ejemplo, si intentamos obtener un usuario que no existe, Spring Boot devuelve una respuesta HTTP con el código de estado 404 (Not Found) y el mensaje de error "User not found".

Tenemos dos opciones para manejar los errores:

- **Manejo de errores con @ExceptionHandler**
- **Manejo de errores con ResponseEntity**

Manejo de errores con @ExceptionHandler

@ExceptionHandler en Spring es una anotación que nos permite manejar excepciones específicas que se lanzan en un controlador de Spring. Esta anotación se utiliza para capturar excepciones específicas y proporcionar una respuesta personalizada al cliente, en lugar de simplemente lanzar la excepción y mostrar el mensaje de error predeterminado.

Podemos la anotación @ControllerAdvice para crear un manejador de excepciones global para nuestra aplicación. Dentro de este manejador, podemos crear un método que se ejecutará cuando se produzca una excepción de tipo 'MiExcepcion'.

Un ejemplo de manejo de errores:

```
@ControllerAdvice
public class ManejadorExcepciones {

    @ExceptionHandler(MiExcepcion.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ResponseBody
    public String manejarMiExcepcion(MiExcepcion ex) {
        return "Ocurrió un error al procesar la solicitud: " + ex.getMessage();
    }
}

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

```
    //Obtener un usuario por su ID
    @GetMapping("/{id}")
    public User getUserById(@PathVariable int id) {
        if (id == 1) {
            throw new MiExcepcion("El usuario con ID 1 no existe");
        }
        return users.stream()
                .filter(user -> user.getId() == id)
                .findFirst()
                .orElse(null);
    }
}
```

Manejo de errores con ResponseEntity

ResponseEntity en Spring es una clase que representa la respuesta HTTP que se envía al cliente. Esta clase es muy útil cuando queremos personalizar la respuesta que se envía al cliente, ya que nos permite configurar el código de estado HTTP, las cabeceras y el cuerpo de la respuesta.

La forma más común de utilizar ResponseEntity es devolverla desde un controlador de Spring. Por ejemplo, supongamos que tenemos un controlador que recibe una solicitud HTTP POST para crear un nuevo recurso. Si la solicitud es válida, el controlador crea el recurso y devuelve una respuesta HTTP con el código de estado 201 (Created) y el recurso creado en el cuerpo de la respuesta. Si la solicitud no es válida, el controlador devuelve una respuesta HTTP con el código de estado 400 (Bad Request) y el mensaje de error en el cuerpo de la respuesta.

Un ejemplo de manejo de errores:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;

@RestController
@RequestMapping("/api")
public class EjemploController {

    @PostMapping("/ejemplo")
    public ResponseEntity<String> crearEjemplo(@Valid @RequestBody EjemploDto
ejemploDto, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            String mensajeError = bindingResult.getFieldErrors().stream()
                    .map(error -> error.getField() + " " + error.getDefaultMessage())
                    .reduce("", (acumulado, mensaje) -> acumulado + mensaje + ", ");
            mensajeError = mensajeError.substring(0, mensajeError.length() - 2); //
eliminamos la última coma y el espacio
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Error de
validación: " + mensajeError);
```

```
        }

        // Código para guardar el ejemplo en la base de datos o realizar cualquier
otra operación

        return ResponseEntity.status(HttpStatus.CREATED).body("Ejemplo creado con
éxito");
    }
}
```

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.

Tipos de acceso a los datos en Spring Data

- **JPQL**: Java Persistence Query Language (JPQL) es un lenguaje de consulta orientado a objetos que se utiliza para realizar consultas en bases de datos relacionales. JPQL es similar a SQL, pero está orientado a objetos y utiliza nombres de clases y atributos en lugar de nombres de tablas y columnas. JPQL se utiliza para realizar consultas en bases de datos relacionales que utilizan el estándar JPA (Java Persistence API) para el mapeo objeto-relacional (ORM).
- **JPA**: Java Persistence API (JPA) es una especificación de Java que define cómo acceder a los datos de una base de datos relacional. JPA se utiliza para mapear objetos Java a tablas de una base de datos relacionales y viceversa. JPA proporciona una API para realizar operaciones CRUD (crear, leer, actualizar y eliminar) en una base de datos relacionales.

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.

JPA

Data JPA es una biblioteca o framework de Java que proporciona una abstracción de alto nivel para interactuar con bases de datos relacionales utilizando la tecnología Java Persistence API (JPA). JPA es una especificación de Java EE para el mapeo objeto-relacional (ORM), que permite a los desarrolladores trabajar con bases de datos relacionales utilizando objetos Java en lugar de SQL directamente.

Data JPA agrega una capa de abstracción adicional en la parte superior de JPA para facilitar la interacción con las bases de datos y reducir la cantidad de código que se debe escribir para realizar operaciones de base de datos. Proporciona características como la creación automática de consultas a partir de métodos de repositorio, la gestión de transacciones y la especificación de consultas

personalizadas.

Data JPA se puede utilizar en combinación con cualquier implementación de JPA, como Hibernate o EclipseLink, y es compatible con una variedad de bases de datos relacionales como MySQL, PostgreSQL, Oracle y SQL Server, entre otras.

Realizar consultas con JPA

Para realizar consultas con JPA, debes crear una interfaz que extienda la interfaz CrudRepository. Esta interfaz proporciona una serie de métodos para realizar operaciones CRUD (crear, leer, actualizar y eliminar) en una base de datos.

Los métodos de la interfaz CrudRepository:

- **save**: Este método se utiliza para guardar un objeto en la base de datos. Si el objeto ya existe en la base de datos, se actualizará. Si el objeto no existe en la base de datos, se creará un nuevo registro.
- **findById**: Este método se utiliza para buscar un objeto en la base de datos utilizando su clave primaria.
- **findAll**: Este método se utiliza para recuperar todos los objetos de una tabla.
- **deleteById**: Este método se utiliza para eliminar un objeto de la base de datos utilizando su clave primaria.
- **delete**: Este método se utiliza para eliminar un objeto de la base de datos.

Los métodos de búsqueda personalizados:

- **findByNombre**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre.
- **findByNombreAndApellido**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y apellido.
- **findByNombreOrApellido**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre o apellido.
- **findByNombreOrderByApellido**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y ordenar los resultados por el atributo apellido.
- **findByNombreNot**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y excluyendo los resultados que coincidan con el atributo nombre.
- **findByNombreLike**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y buscando los resultados que coincidan con el atributo nombre.
- **findByNombreStartingWith**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y buscando los resultados que coincidan con el atributo nombre al comienzo de la cadena.
- **findByNombreEndingWith**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y buscando los resultados que coincidan con el atributo nombre al final de la cadena.
- **findByNombreContaining**: Este método se utiliza para buscar un objeto en la base de datos

utilizando el atributo nombre y buscando los resultados que coincidan con el atributo nombre en cualquier parte de la cadena.

- **findByNombreIn**: Este método se utiliza para buscar un objeto en la base de datos utilizando el atributo nombre y buscando los resultados que coincidan con el atributo nombre en una lista de valores.

Para implementar consultas con JPA:

- Crea una interfaz que extienda la interfaz JpaRepository.
- Anota la interfaz con @Repository para indicar que es un repositorio.
- Crea un método que devuelva un objeto de la entidad que se desea buscar.

Definiendo un interfaz de repositorio con varios métodos de búsqueda personalizados:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(String email);
    User findByEmailAndName(String email, String name);
    User findByEmailOrName(String email, String name);
    User findByEmailOrderByLastName(String email);
    User findByEmailNot(String email);
}
```

La clase del servicio:

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User findByEmail(String email) {
        return userRepository.findByEmail(email);
    }

    @Transactional
    public addUser(User user) {
        userRepository.save(user);
    }
}
```

JPQL

las consultas JPQL permiten realizar peticiones orientadas a objetos en una base de datos relacional utilizando JPA. A diferencia de SQL, las consultas JPQL se realizan sobre entidades y sus atributos, lo que permite a los desarrolladores trabajar con objetos Java directamente.

Aquí hay algunos conceptos clave a tener en cuenta al trabajar con consultas JPQL:

- **Entidades**: Las entidades son clases Java que representan tablas de base de datos. Las consultas

JPQL se realizan sobre entidades.

- **Atributos**: Los atributos son variables dentro de una entidad que representan columnas en la base de datos. *Clausulas: Las cláusulas son palabras clave utilizadas en una consulta para definir lo que se busca. Las cláusulas comunes incluyen SELECT, FROM, WHERE, ORDER BY, y GROUP BY.
- **Parámetros**: Los parámetros se utilizan para proporcionar valores dinámicos a las consultas. Los parámetros se indican en la consulta utilizando la sintaxis ":" seguida del nombre del parámetro.
- **Funciones**: Las funciones son expresiones que se aplican a los atributos de las entidades para realizar cálculos o manipulaciones de datos. Las funciones comunes incluyen AVG, MAX, MIN, COUNT, y SUM.
- **Resultados**: Los resultados de las consultas JPQL son objetos Java que se pueden utilizar en el código de la aplicación.

Las anotaciones de JPQL:

- **@Query**: Anota un método de un repositorio para indicar que se debe ejecutar una consulta JPQL. Esta anotación se utiliza para definir consultas JPQL personalizadas.
- **@Param**: Anota un parámetro de un método de un repositorio para indicar que se debe utilizar como parámetro en una consulta JPQL. Esta anotación se utiliza para definir consultas JPQL personalizadas.
- **@Modifying**: Anota un método de un repositorio para indicar que se debe ejecutar una consulta JPQL que modifica los datos. Esta anotación se utiliza para definir consultas JPQL personalizadas.

Ejemplo de consulta JPQL:

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

Ejemplo de consulta JPQL con @Modifying:

```
public interface ProductoRepository extends JpaRepository<Producto, Long> {
    @Modifying
    @Query("UPDATE Producto p SET p.precio = :precio WHERE p.id = :id")
    void actualizar(@Param("id") Long id, @Param("precio") Double precio);
}
```

Un ejemplo de un servicio que utiliza un repositorio para realizar consultas JPQL:

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

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

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

Hibernate

Hibernate es un framework de mapeo objeto-relacional (ORM) que se utiliza comúnmente en aplicaciones de Java. Es una implementación de la especificación Java Persistence API (JPA) y proporciona una forma de interactuar con bases de datos relacionales utilizando objetos Java.

En el contexto de Spring, Hibernate se utiliza a menudo como implementación de JPA para acceder a bases de datos relacionales. Spring proporciona una capa adicional de abstracción sobre JPA y Hibernate, facilitando su integración en una aplicación de Spring.

Para utilizar Hibernate con Spring, se debe configurar un EntityManagerFactory, que es una fábrica de objetos EntityManager. EntityManager es una interfaz de JPA que permite realizar operaciones de lectura y escritura en la base de datos utilizando objetos Java.

Para establecer relaciones entre entidades en Spring con Hibernate, se utilizan las siguientes anotaciones:

- **@OneToOne**: Esta anotación se utiliza para establecer una relación uno-a-uno entre dos entidades. Se utiliza para indicar que una entidad tiene una relación con otra entidad de forma que una instancia de una entidad sólo puede estar asociada con una única instancia de la otra entidad.
- **@OneToMany**: Esta anotación se utiliza para establecer una relación uno-a-muchos entre dos entidades. Se utiliza para indicar que una entidad tiene una colección de instancias de otra entidad.
- **@ManyToOne**: Esta anotación se utiliza para establecer una relación muchos-a-uno entre dos entidades. Se utiliza para indicar que varias instancias de una entidad pueden estar asociadas con una única instancia de otra entidad.
- **@ManyToMany**: Esta anotación se utiliza para establecer una relación muchos-a-muchos entre dos entidades. Se utiliza para indicar que varias instancias de una entidad pueden estar asociadas con varias instancias de otra entidad.

Cada una de estas anotaciones se utiliza en combinación con otras anotaciones de Hibernate, como **@JoinColumn**, **@JoinTable** o **@MappedBy** , para especificar la forma en que se deben mapear las relaciones entre las entidades.

Ejemplo de uso de @OneToOne:

```
@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;

    @OneToOne
    @JoinColumn(name = "category_id")
    private Categoria categoria;

    // constructor, getters y setters
}

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

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

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

    // constructor, getters y setters
}
```

Ejemplo de uso de @OneToMany:

```
@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;

    @OneToMany(mappedBy = "producto")
    private List<DetallePedido> detallesPedido;

    // constructor, getters y setters
}

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

    @Column(name = "quantity")
    private Integer cantidad;

    @ManyToOne
    @JoinColumn(name = "product_id")
    private Producto producto;

    @ManyToOne
    @JoinColumn(name = "order_id")
    private Pedido pedido;

    // constructor, getters y setters
}
```

Ejemplo de uso de @ManyToOne:

```
@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;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Categoria categoria;

    // constructor, getters y setters
```

```
}

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

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

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

    @OneToMany(mappedBy = "categoria")
    private List<Producto> productos;

    // constructor, getters y setters
}@
@
@
```

Ejemplo de uso de @ManyToMany:

```
@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;

    @ManyToMany
    @JoinTable(name = "product_tag",
            joinColumns = @JoinColumn(name = "product_id"),
            inverseJoinColumns = @JoinColumn(name = "tag_id"))
    private List<Etiqueta> etiquetas;

    // constructor, getters y setters
}

@Entity
public class Etiqueta {
```

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

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

    @ManyToMany(mappedBy = "etiquetas")
    private List<Producto> productos;

    // constructor, getters y setters
}
```

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>
```

Thymeleaf

Thymeleaf es un motor de plantillas para aplicaciones web en Java que permite integrar HTML, CSS y JavaScript con datos dinámicos que se generan en el lado del servidor. Se utiliza principalmente en aplicaciones web basadas en el framework Spring.

Thymeleaf es relativamente flexible, y su sintaxis es muy similar a HTML, lo que lo hace muy fácil de aprender. Una de las principales ventajas de Thymeleaf es su capacidad para procesar plantillas tanto en el lado del servidor como en el lado del cliente.

En el lado del servidor, Thymeleaf puede procesar plantillas HTML con marcadores de posición para variables y expresiones, que se reemplazan por valores dinámicos en el servidor antes de enviar la respuesta HTTP al cliente. En el lado del cliente, Thymeleaf puede procesar plantillas HTML que se han enviado desde el servidor y actualizan el contenido dinámico en la página sin necesidad de realizar una nueva solicitud HTTP.

Las directivas de Thymeleaf

Las directivas de Thymeleaf son:

- **th:text**: permite mostrar el valor de una expresión o variable en el contenido de un elemento HTML.
- **th:if** y th:unless: permite incluir o excluir contenido HTML basado en una condición booleana.
- **th:switch** y th:case: permite realizar una selección de casos basada en una expresión y mostrar un contenido HTML diferente para cada caso.
- **th:each**: permite iterar sobre una colección de objetos y mostrar un contenido HTML para cada elemento.
- **th:href** y th:src: permite definir la URL de un enlace o de una imagen de manera dinámica a través de una expresión.
- **th:object**: permite establecer un objeto como contexto para la evaluación de expresiones dentro de un fragmento HTML.
- **th:fragment**: permite definir un fragmento HTML que puede ser incluido en otras plantillas a través de la directiva th:include.
- **th:include**: permite incluir un fragmento HTML definido en otra plantilla.
- **th:attr**: permite agregar atributos HTML dinámicamente a un elemento a través de una expresión.
- **th:value**: permite establecer el valor de un atributo HTML a través de una expresión.
- **th:checked**: permite establecer el estado de una casilla de verificación o de un botón de opción a través de una expresión.
- **th:style**: permite establecer dinámicamente el valor de una regla de estilo CSS.

Un ejemplo de uso de th:each:

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Descripción</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${productos}">
      <td th:text="${prod.nombre}">Nombre del producto</td>
      <td th:text="${prod.descripcion}">Descripción del producto</td>
      <td th:text="${prod.precio}">Precio del producto</td>
```

```
    </tr>
  </tbody>
</table>
```

Un ejemplo de uso de th:if:

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Descripción</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${productos}" th:if="${prod.precio > 10}">
      <td th:text="${prod.nombre}">Nombre del producto</td>
      <td th:text="${prod.descripcion}">Descripción del producto</td>
      <td th:text="${prod.precio}">Precio del producto</td>
    </tr>
  </tbody>
</table>
```

Un ejemplo de uso de th:switch:

```
<table>
    <thead>
        <tr>
        <th>Nombre</th>
        <th>Descripción</th>
        <th>Precio</th>https://www.baeldung.com/spring-boot-testing
        <th>Tipo</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="prod : ${productos}">
        <td th:text="${prod.nombre}">Nombre del producto</td>
        <td th:text="${prod.descripcion}">Descripción del producto</td>
            <td th:text="${prod.precio}">Precio del producto</td>
            <td th:switch="${prod.tipo}">
                <span th:case="'A'">Tipo A</span>
                <span th:case="'B'">Tipo B</span>
                <span th:case="'C'">Tipo C</span>
                <span th:case="*">Tipo desconocido</span>
            </td>
        </tr>
    </tbody>
</table>
```

Un ejemplo de uso de th:href:

```
<a th:href="@{/productos/{id}(id=${prod.id})}">Ver detalles</a>
```

Un ejemplo de uso de th:object:

```
<div th:object="${producto}">
    <p>Nombre: <span th:text="*{nombre}">Nombre del producto</span></p>
    <p>Descripción: <span th:text="*{descripcion}">Descripción del producto</span></p>
    <p>Precio: <span th:text="*{precio}">Precio del producto</span></p>
</div>
```

Un ejemplo de uso de th:fragment:

```
<div th:fragment="producto">
    <p>Nombre: <span th:text="*{nombre}">Nombre del producto</span></p>
    <p>Descripción: <span th:text="*{descripcion}">Descripción del producto</span></p>
    <p>Precio: <span th:text="*{precio}">Precio del producto</span></p>
</div>
```

Spring Security

¿Qué es Spring Security?

Spring Security es un framework de seguridad que proporciona características de autenticación, autorización y protección contra ataques de seguridad en aplicaciones basadas en Spring.

Al utilizar Spring Security, podemos asegurarnos de que los usuarios que acceden a nuestra aplicación estén autenticados y autorizados para realizar determinadas acciones. También nos permite proteger nuestras aplicaciones contra ataques comunes como Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS) y SQL Injection.

Spring Security se basa en filtros y proveedores de autenticación y autorización, que se encargan de procesar las solicitudes de los usuarios y verificar su identidad y permisos.

Algunos de los componentes principales de Spring Security son:

- **UserDetailsService**: es una interfaz que se utiliza para cargar la información de los usuarios y sus roles desde una fuente de datos, como una base de datos o un servicio web.
- **AuthenticationManager**: es una interfaz que se encarga de procesar las solicitudes de autenticación y validar las credenciales del usuario.
- **PasswordEncoder**: es una interfaz que se utiliza para codificar y decodificar las contraseñas de los usuarios, para que no se almacenen en texto plano en la base de datos o en otros medios de almacenamiento.
- **AccessDecisionManager**: es una interfaz que se utiliza para tomar decisiones de autorización y determinar si un usuario tiene acceso a una determinada funcionalidad o recurso en la

aplicación.

Para utilizar Spring Security en una aplicación, primero debemos agregar las dependencias correspondientes en el archivo pom.xml o build.gradle, según sea el caso. Luego, podemos configurar las reglas de seguridad y las opciones de autenticación y autorización en el archivo application.properties o mediante clases de configuración específicas de Spring Security.

Las dependencias de Spring Security en Maven son:

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

Las dependencias de Spring Security en Gradle son:

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

Spring Security también proporciona integración con otros frameworks y herramientas de seguridad, como OAuth2, JWT, LDAP y SAML, lo que nos permite implementar características avanzadas de seguridad en nuestras aplicaciones de manera sencilla y eficiente.

En resumen, Spring Security es una herramienta esencial para asegurar nuestras aplicaciones Spring y protegerlas contra vulnerabilidades y ataques de seguridad.

UserDetailsService

En Spring Security, UserDetailsService es una interfaz que se utiliza para cargar la información de los usuarios y sus roles desde una fuente de datos, como una base de datos o un servicio web.

Cuando un usuario intenta autenticarse en nuestra aplicación, Spring Security utiliza el UserDetailsService para cargar la información del usuario y verificar sus credenciales. Si las credenciales son válidas, Spring Security crea un objeto Authentication con la información del usuario y lo almacena en el contexto de seguridad de la aplicación.

Para implementar un UserDetailsService, debemos crear una clase que implemente la interfaz y sobrescribir el método loadUserByUsername, que se utiliza para cargar la información del usuario a partir de su nombre de usuario.

En este método, podemos realizar consultas a la base de datos o a cualquier otro servicio de autenticación externo para obtener la información del usuario. La información del usuario debe ser devuelta en forma de un objeto UserDetails, que contiene información como el nombre de usuario, la contraseña, los roles y los permisos del usuario.

Aquí hay un ejemplo básico de cómo implementar un UserDetailsService en Spring Security:

```
@Service
```

```
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }
        return new CustomUserDetails(user);
    }
}
```

En este ejemplo, estamos cargando la información del usuario a partir de una base de datos utilizando el UserRepository, que es una interfaz que se encarga de las operaciones de CRUD en la tabla de usuarios. Después de cargar la información del usuario, estamos devolviendo un objeto CustomUserDetails, que es una clase que implementa la interfaz UserDetails y contiene la información del usuario y sus roles.

Aquí hay un ejemplo de cómo implementar un UserDetailsService en Spring Security que utiliza una base de datos para obtener la información del usuario:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }
        return new org.springframework.security.core.userdetails.User(
                user.getUsername(), user.getPassword(), getAuthorities(user));
    }

    private Collection<? extends GrantedAuthority> getAuthorities(User user) {
        String[] userRoles = user.getRoles().stream().map((role) ->
role.getName()).toArray(String[]::new);
        Collection<GrantedAuthority> authorities =
AuthorityUtils.createAuthorityList(userRoles);
        return authorities;
    }
```

```
}
```

AuthenticationManager

AuthenticationManager es una clase en el framework de seguridad de Spring que se encarga de manejar la autenticación en una aplicación web. Es responsable de tomar las credenciales del usuario, autenticarlas y crear un objeto de autenticación para el usuario si se ha autenticado correctamente.

El proceso de autenticación puede variar dependiendo de la configuración de la aplicación, pero generalmente sigue los siguientes pasos:

El usuario proporciona sus credenciales, como un nombre de usuario y una contraseña. El sistema valida las credenciales, generalmente mediante una comparación con información almacenada en una base de datos. Si las credenciales son válidas, se crea un objeto de autenticación para el usuario, que contiene detalles como el nombre de usuario, los roles y cualquier otra información relevante. El objeto de autenticación se almacena en el contexto de seguridad de la aplicación para que se pueda acceder posteriormente. Aquí hay un ejemplo de cómo se puede usar AuthenticationManager en una aplicación Spring Boot:

Supongamos que tenemos una entidad "Usuario" en nuestra aplicación, que tiene propiedades como "nombre de usuario" y "contraseña". Además, tenemos una clase "UserService" que se encarga de interactuar con la base de datos y realizar la validación de credenciales.

Primero, definimos una configuración de seguridad básica en nuestra clase principal:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserService userService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasRole("USER")
            .and().formLogin()
            .and().logout()
            .logoutSuccessUrl("/login?logout=true")
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .deleteCookies("JSESSIONID");
```

```
    }
}
```

Luego, en nuestra clase **UserService***, definimos el método* **loadUserByUsername** *para buscar al usuario en la base de datos y validar sus credenciales:*

```
@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("Usuario no encontrado");
        }
        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                user.isEnabled(),
                true,
                true,
                true,
                getAuthorities(user.getRoles()));
    }

    private Collection<? extends GrantedAuthority> getAuthorities(
            Collection<Role> roles) {
        return roles.stream()
                .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName()))
                .collect(Collectors.toList());
    }
}
```

AuthenticationProvider

AuthenticationProvider es una interfaz en Spring Security que se utiliza para autenticar solicitudes de seguridad en una aplicación web. Esta interfaz se encarga de validar las credenciales de un usuario y autenticarlo en la aplicación.

AuthenticationProvider se utiliza en conjunto con AuthenticationManager para proporcionar la lógica de autenticación. AuthenticationManager utiliza uno o más proveedores de autenticación (AuthenticationProvider) para realizar la autenticación.

La interfaz AuthenticationProvider tiene un único método llamado "authenticate", que toma como parámetro un objeto Authentication y devuelve un objeto Authentication si la autenticación es

exitosa. Si la autenticación falla, el método lanza una excepción AuthenticationException.

Un ejemplo de implementación de un AuthenticationProvider en Spring Security sería el siguiente:

```
public class MyAuthenticationProvider implements AuthenticationProvider {

  @Autowired
  private UserService userService;

  @Override
  public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
    String username = authentication.getName();
    String password = authentication.getCredentials().toString();

    User user = userService.findByUsername(username);

    if (user == null) {
      throw new BadCredentialsException("Usuario no encontrado");
    }

    if (!passwordEncoder.matches(password, user.getPassword())) {
      throw new BadCredentialsException("Contraseña incorrecta");
    }

    List<GrantedAuthority> roles = new ArrayList<>();
    roles.add(new SimpleGrantedAuthority(user.getRole().getName()));

    return new UsernamePasswordAuthenticationToken(username, password, roles);
  }

  @Override
  public boolean supports(Class<?> authentication) {
    return authentication.equals(UsernamePasswordAuthenticationToken.class);
  }
}
```

PasswordEncoder

PasswordEncoder en Spring Boot es una interfaz que se utiliza para cifrar y descifrar contraseñas. El objetivo de cifrar contraseñas es evitar que las contraseñas en texto plano sean almacenadas en una base de datos, lo que puede ser un problema de seguridad. En su lugar, se almacena el valor cifrado de la contraseña en la base de datos.

Spring Boot proporciona varias implementaciones de la interfaz PasswordEncoder. Una de las implementaciones más comunes es BCryptPasswordEncoder. BCryptPasswordEncoder utiliza el algoritmo bcrypt para cifrar contraseñas. El algoritmo bcrypt es un algoritmo de cifrado de contraseñas sólido y seguro, que utiliza una técnica de "salting" (añadir una cadena aleatoria) para hacer que el cifrado sea más fuerte.

Para utilizar PasswordEncoder en Spring Boot, primero debes crear una instancia de una implementación de la interfaz PasswordEncoder.

Para usar, hay que importar la dependencia de Spring Security:

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

Luego, en nuestra clase principal, podemos inyectar una instancia de PasswordEncoder en nuestro código:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

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

Ahora, puedes utilizar la instancia de PasswordEncoder en tus servicios de usuario (por ejemplo, UserService). En el siguiente ejemplo, utilizamos PasswordEncoder para cifrar la contraseña antes de almacenarla en la base de datos:

```
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public UserService(UserRepository userRepository, PasswordEncoder passwordEncoder)
{
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    public void saveUser(User user) {
        String encodedPassword = passwordEncoder.encode(user.getPassword());
        user.setPassword(encodedPassword);
        userRepository.save(user);
```

```
    }
}
```

Por último, puedes utilizar PasswordEncoder en tus controladores de Spring. En el siguiente ejemplo, utilizamos PasswordEncoder para verificar que la contraseña proporcionada por el usuario coincide con la contraseña almacenada en la base de datos:

```
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    private final UserService userService;
    private final PasswordEncoder passwordEncoder;

    public UserController(UserService userService, PasswordEncoder passwordEncoder) {
        this.userService = userService;
        this.passwordEncoder = passwordEncoder;
    }

    @PostMapping("/login")
    public void login(@RequestBody LoginRequest loginRequest) {
        User user = userService.findUserByUsername(loginRequest.getUsername());

        if (user != null) {
            if (passwordEncoder.matches(loginRequest.getPassword(),
user.getPassword())) {
                // la contraseña coincide, inicia sesión
            } else {
                // la contraseña no coincide, muestra un mensaje de error
            }
        } else {
            // el usuario no existe, muestra un mensaje de error
        }
    }
}
```

Un curl que nos permite probar el login:

```
curl -X POST -H "Content-Type: application/json" -d
'{"username":"usuario","password":"contraseña"}' http://localhost:8080/login
```

AccessDecisionManager

AccessDecisionManager es una interfaz que se utiliza para tomar decisiones de acceso a los

recursos protegidos. Su función principal es determinar si un usuario tiene los permisos necesarios para acceder a un recurso determinado.

AccessDecisionManager trabaja en conjunto con otros componentes de Spring Security, como los filtros de seguridad y los proveedores de autenticación, para determinar si un usuario puede acceder a un recurso. En particular, AccessDecisionManager se utiliza para tomar decisiones de acceso en el nivel de autorización después de que un usuario ha sido autenticado.

AccessDecisionManager trabaja en conjunto con otros componentes de Spring Security, como los filtros de seguridad y los proveedores de autenticación, para determinar si un usuario puede acceder a un recurso. En particular, AccessDecisionManager se utiliza para tomar decisiones de acceso en el nivel de autorización después de que un usuario ha sido autenticado.

Para tomar una decisión de acceso, AccessDecisionManager toma en cuenta varios factores, incluyendo los siguientes:

- La identidad del usuario que intenta acceder al recurso
- Los permisos y roles asociados con el usuario
- La configuración de seguridad del recurso que se está intentando acceder

AccessDecisionManager puede tomar una de tres decisiones posibles:

- Permitir el acceso al recurso: esto significa que el usuario tiene los permisos necesarios para acceder al recurso y se le permitirá hacerlo.
- Denegar el acceso al recurso: esto significa que el usuario no tiene los permisos necesarios para acceder al recurso y se le denegará el acceso.
- Delegar la decisión a un componente diferente: esto significa que AccessDecisionManager no puede tomar una decisión definitiva y delegará la decisión a otro componente.

La implementación predeterminada de AccessDecisionManager en Spring Security es AffirmativeBased, que sigue una lógica de "permiso concedido" para tomar decisiones de acceso. Esto significa que, en general, si un usuario tiene al menos un permiso que le permite acceder a un recurso determinado, se le permitirá el acceso.

Para utilizar AccessDecisionManager en Spring Boot, primero debes crear una instancia de una implementación de la interfaz AccessDecisionManager. En el siguiente ejemplo:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/admin").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .httpBasic();
```

```
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("usuario").password("contraseña").roles("USER")
            .and()
            .withUser("admin").password("contraseña").roles("ADMIN");
    }

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

    @Bean
    public AccessDecisionManager accessDecisionManager() {
        return new AffirmativeBased(Arrays.asList(new RoleVoter()));
    }
}
```

Aquí, estamos configurando la seguridad de nuestra aplicación de la siguiente manera:

- "/admin" está protegido y solo los usuarios con el rol "ADMIN" pueden acceder a él.
- Cualquier otra solicitud debe estar autenticada.
- Estamos utilizando la autenticación en memoria, pero en una aplicación real, deberíamos utilizar un proveedor de autenticación más robusto.
- Estamos usando un codificador de contraseñas BCryptPasswordEncoder para codificar las contraseñas.
- Estamos creando un AccessDecisionManager usando la implementación AffirmativeBased y un RoleVoter.

Casos de uso de Spring Security

Autenticación básica

La autenticación básica es un método de autenticación simple que se utiliza para proteger los recursos de una aplicación web. En este método, el cliente envía las credenciales de autenticación (nombre de usuario y contraseña) en cada solicitud HTTP. El servidor verifica las credenciales de autenticación y, si son válidas, devuelve el recurso solicitado.

Las dependencias necesarias para utilizar la autenticación básica en Spring Boot son las siguientes:

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

Para utilizar la autenticación básica en Spring Boot, primero debes crear una instancia de la clase HttpBasicConfigurer. En el siguiente ejemplo:

```
@Configuration
@EnableWebSecurity
public class CustomWebSecurityConfigurerAdapter {

    @Autowired private MyBasicAuthenticationEntryPoint authenticationEntryPoint;

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
          .inMemoryAuthentication()
          .withUser("user1")
          .password(passwordEncoder().encode("user1Pass"))
          .authorities("ROLE_USER");
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/securityNone")
            .permitAll()
            .anyRequest()
            .authenticated()
            .and()
            .httpBasic()
            .authenticationEntryPoint(authenticationEntryPoint);
        http.addFilterAfter(new CustomFilter(), BasicAuthenticationFilter.class);
        return http.build();
    }

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

Para probarlo, podemos usar curl:

```
curl -i --user user1:user1Pass http://localhost:8080/spring-security-rest-basic-
auth/api/foos/1
```

Por defecto, el BasicAuthenticationEntryPoint proporcionado por Spring Security devuelve una página completa para una respuesta 401 Unauthorized de vuelta al cliente. Esta representación HTML del error se muestra bien en un navegador. Por el contrario, no está bien adaptado para otros escenarios, como una API REST donde se puede preferir una representación json.

El nuevo entrypoint se define como un bean estándar:

```
@Component
public class MyBasicAuthenticationEntryPoint extends BasicAuthenticationEntryPoint {

    @Override
    public void commence(
      HttpServletRequest request, HttpServletResponse response,
AuthenticationException authEx)
      throws IOException, ServletException {
        response.addHeader("WWW-Authenticate", "Basic realm="" + getRealmName() +
""");
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        PrintWriter writer = response.getWriter();
        writer.println("HTTP Status 401 - " + authEx.getMessage());
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        setRealmName("Baeldung");
        super.afterPropertiesSet();
    }
}
```

Manejo de sesiones

En una aplicación Spring Boot, el manejo de sesiones se puede realizar de varias maneras. Una de las formas más comunes de manejar las sesiones es a través del uso de cookies de sesión.

Cuando un usuario inicia sesión en una aplicación web, se crea una sesión en el servidor que contiene la información de la sesión, como el ID de sesión y cualquier dato adicional que se haya almacenado en la sesión. El servidor envía una cookie de sesión al navegador del usuario, que contiene el ID de sesión. El navegador envía la cookie de sesión en cada solicitud posterior, permitiendo que el servidor identifique la sesión del usuario.

Las dependencias necesarias para utilizar el manejo de sesiones en Spring Boot son las siguientes:

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

Para utilizar el manejo de sesiones en Spring Boot, primero debes crear una instancia de la clase HttpSessionConfigurer. En el siguiente ejemplo:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/login").permitAll()
                .anyRequest().authenticated()
            .and()
            .formLogin()
                .loginPage("/login")
                .defaultSuccessUrl("/home")
                .permitAll()
            .and()
            .logout()
                .logoutUrl("/logout")
                .permitAll()
            .and()
            .sessionManagement()
                .invalidSessionUrl("/login?expired")
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true);
    }
}
```

El controlador de inicio de sesión se define de la siguiente manera:

```
@Controller
public class HomeController {

    @GetMapping("/login")
    public String showLoginForm(Model model) {
        return "login";
    }

    @GetMapping("/home")
    public String showHomePage() {
        return "home";
    }

    @GetMapping("/logout")
    public String logout(HttpServletRequest request) throws ServletException {
        request.logout();
        return "redirect:/login?logout";
    }

}
```

login.html

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

```
<head>
    <title>Login</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <h1>Login</h1>
    <div th:if="${param.error}">
        <p>Invalid username and password.</p>
    </div>
    <div th:if="${param.logout}">
        <p>You have been logged out.</p>
    </div>
    <form th:action="@{/login}" method="post">
        <div>
            <label for="username">Username:</label>
            <input type="text" id="username" name="username" autofocus required>
        </div>
        <div>
            <label for="password">Password:</label>
            <input type="password" id="password" name="password" required>
        </div>
        <div>
            <button type="submit">Login</button>
        </div>
    </form>
</body>
</html>
```

home.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Home</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <h1>Welcome!</h1>
    <p>You are now logged in.</p>
    <form th:action="@{/logout}" method="post">
        <button type="submit">Logout</button>
    </form>
</body>
</html>
```

Beans de sesión (Session Scoped Beans)

En Spring Framework, un bean con ámbito de sesión (Session Scoped Bean) es un objeto que está asociado a una sesión HTTP individual. Esto significa que una instancia de bean se crea y se almacena en la sesión HTTP del usuario cuando se crea una nueva sesión, y se destruye cuando la sesión termina. Los beans con ámbito de sesión son útiles cuando se necesita mantener el estado entre múltiples solicitudes de un mismo usuario.

Las dependencias necesarias para utilizar los beans de sesión en Spring Boot son las siguientes:

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

Para utilizar los beans de sesión en Spring Boot, primero hay que crear una clase de configuración que contenga el bean de sesión. En el siguiente ejemplo:

```
@Configuration
@ComponentScan(basePackages = {"com.example.demo"})
public class AppConfig {
    // ...
}
```

Para crear un bean de sesión, debemos utilizar la anotación @SessionScope. En el siguiente ejemplo:

```
@Component
@SessionScope
public class SessionBean {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Para utilizar el bean de sesión, debemos inyectarlo en el controlador. En el siguiente ejemplo:

```
@Controller
public class HomeController {

    @Autowired
    private SessionBean sessionBean;

    @GetMapping("/")
```

```
    public String showHomePage(Model model) {
        model.addAttribute("name", sessionBean.getName());
        return "home";
    }

    @PostMapping("/save")
    public String saveName(@RequestParam String name) {
        sessionBean.setName(name);
        return "redirect:/";
    }

}
```

Obtener información del usuario en Spring Security

Spring Security proporciona una forma de obtener información del usuario actualmente autenticado en la aplicación. Esta información se puede utilizar para realizar acciones específicas para el usuario actualmente autenticado.

Las dependencias necesarias para obtener información del usuario en Spring Security son las siguientes:

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

Configuramos la seguridad de nuestra aplicación de la siguiente manera:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/").permitAll()
                .antMatchers("/secured").authenticated()
            .and()
            .formLogin();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
```

```
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER");
    }
}
```

Creamos un controlador para probar la funcionalidad:

```
@RestController
public class SecuredController {

    @GetMapping("/secured")
    public String securedEndpoint() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String username = authentication.getName();
        return "Hello, " + username + "!";
    }
}
```

Para probarlo, podemos usar curl:

```
curl -i -X GET http://localhost:8080/secured
```

Testing con Spring Boot

En Spring Boot, el testing se puede realizar mediante el uso del framework de pruebas JUnit, que es muy popular en la comunidad de desarrollo de Java. Spring Boot también proporciona una serie de herramientas y clases de ayuda para facilitar la escritura de pruebas de unidad, integración y end-to-end.

En Spring Boot, hay varias formas de hacer testing, dependiendo de las necesidades y objetivos de la aplicación. A continuación, te presento algunas de las formas más comunes de hacer testing en Spring Boot:

- **Pruebas unitarias**: Son pruebas que se realizan para comprobar el correcto funcionamiento de un método o clase aislada. En Spring Boot, las pruebas unitarias se realizan utilizando JUnit y Mockito.
- **Pruebas de integración**: Son pruebas que se realizan para comprobar el correcto funcionamiento de la interacción entre diferentes componentes de la aplicación. En Spring Boot, las pruebas de integración se realizan utilizando Spring Test Framework, que permite cargar el contexto de la aplicación y realizar pruebas de extremo a extremo.
- **Pruebas de aceptación**: Son pruebas que se realizan para comprobar que la aplicación cumple con los requisitos funcionales y no funcionales definidos por el cliente o el usuario final. En Spring Boot, las pruebas de aceptación se realizan utilizando herramientas de automatización de pruebas, como Cucumber o Selenium.
- **Pruebas de rendimiento**: Son pruebas que se realizan para comprobar el rendimiento de la

aplicación en diferentes escenarios de carga y tráfico. En Spring Boot, las pruebas de rendimiento se realizan utilizando herramientas de benchmarking, como JMeter o Gatling.

Pruebas de seguridad: Son pruebas que se realizan para comprobar la seguridad de la aplicación, detectar vulnerabilidades y asegurar la protección de los datos y la privacidad de los usuarios. En Spring Boot, las pruebas de seguridad se realizan utilizando herramientas de análisis estático y dinámico, como SonarQube o OWASP ZAP.

Pruebas unitarias con JUnit

Aquí hay un ejemplo básico de cómo realizar pruebas de unidad en una aplicación Spring Boot utilizando JUnit:

Supongamos que tienes una clase de servicio llamada "UserService" que contiene un método llamado "getUserById" que recibe un id de usuario y devuelve un objeto User correspondiente a ese id.

```
@Service
public class UserService {

    public User getUserById(Long id) {
        // ...
    }
}
```

Ahora queremos probar que el método "getUserById" funciona correctamente. Aquí está el código de prueba utilizando JUnit:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    public void testGetUserById() {
        User user = userService.getUserById(1L);
        assertNotNull(user);
        assertEquals("John", user.getName());
        assertEquals("Doe", user.getSurname());
    }

}
```

En este ejemplo, estamos utilizando la anotación "@RunWith(SpringRunner.class)" para indicar que queremos ejecutar las pruebas utilizando el contexto de Spring. También utilizamos la anotación "@SpringBootTest" para indicar que queremos cargar el contexto completo de la aplicación Spring Boot para nuestras pruebas.

Luego, en el método de prueba "testGetUserById", utilizamos la anotación "@Autowired" para inyectar la instancia de UserService en la prueba. Luego, llamamos al método "getUserById" y realizamos algunas aserciones para comprobar que el objeto User devuelto por el método es el correcto.

La sintaxis de las pruebas unitarias con JUnit

JUnit es un framework de testing para aplicaciones Java, que permite realizar pruebas unitarias de manera sencilla y eficiente. A continuación, te presento un resumen de la sintaxis básica de JUnit:

- Anotaciones: JUnit utiliza una serie de anotaciones para marcar los métodos de prueba y proporcionar información adicional sobre las pruebas. Las anotaciones más comunes son:
  - @Test: Indica que un método es una prueba unitaria.
  - @Before: Indica que un método debe ejecutarse antes de cada prueba.
  - @After: Indica que un método debe ejecutarse después de cada prueba.
  - @BeforeClass: Indica que un método debe ejecutarse antes de que se ejecuten todas las pruebas en una clase.
  - @AfterClass: Indica que un método debe ejecutarse después de que se ejecuten todas las pruebas en una clase.
- Asserts: JUnit proporciona una serie de métodos Assert para verificar que los resultados de las pruebas sean los esperados. Los métodos más comunes son:
  - assertEquals: Compara dos valores y verifica que sean iguales.
  - assertTrue/assertFalse: Verifica si una condición es verdadera o falsa.
  - assertNull/assertNotNull: Verifica si un objeto es nulo o no nulo.
  - assertSame/assertNotSame: Verifica si dos objetos son iguales o no iguales.
- Ejecución de pruebas: Para ejecutar pruebas en JUnit, se crea una clase de prueba que contiene uno o más métodos anotados con @Test. Luego, se ejecuta la prueba utilizando un runner, como el runner predeterminado de JUnit, que proporciona informes sobre el estado de las pruebas.

Pruebas de integración con Spring Test Framework

En Spring Boot, las pruebas de integración se realizan utilizando el framework Spring Test, que permite cargar el contexto de la aplicación y realizar pruebas de extremo a extremo.

Para realizar pruebas de integración en Spring Boot se crea una clase de prueba que carga el contexto de la aplicación y se configura el entorno de prueba utilizando anotaciones como @TestPropertySource y @ActiveProfiles. Se pueden inyectar dependencias utilizando @Autowired o @MockBean y se realizan pruebas utilizando los métodos de aserción de JUnit o herramientas adicionales como MockMvc.

Crear una clase de prueba: Para realizar pruebas de integración en Spring Boot, se crea una clase de prueba en la que se anota con la anotación @RunWith(SpringRunner.class), que indica que se utilizará el runner de Spring Test.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyIntegrationTest {
   // Aquí van los métodos de prueba
}
```

Cargar el contexto de la aplicación: Para cargar el contexto de la aplicación en la clase de prueba, se utiliza la anotación @SpringBootTest, que indica que se debe cargar la configuración completa de la aplicación.

```
@SpringBootTest
public class MyIntegrationTest {
   // Aquí van los métodos de prueba
}
```

Configurar el entorno de prueba: Para configurar el entorno de prueba, se pueden utilizar las anotaciones @TestPropertySource y @ActiveProfiles. La anotación @TestPropertySource permite cargar propiedades específicas para la prueba, mientras que la anotación @ActiveProfiles permite activar perfiles específicos de la aplicación.

```
@SpringBootTest
@TestPropertySource(locations="classpath:test.properties")
@ActiveProfiles("test")
public class MyIntegrationTest {
   // Aquí van los métodos de prueba
}
```

Inyectar dependencias: Para acceder a los componentes de la aplicación, se pueden inyectar dependencias utilizando la anotación @Autowired o @MockBean. La anotación @Autowired permite inyectar una instancia real del componente, mientras que la anotación @MockBean permite inyectar un objeto simulado para pruebas.

```
@SpringBootTest
public class MyIntegrationTest {
   @Autowired
   private MyService myService;

   @MockBean
   private MyRepository myRepository;

   // Aquí van los métodos de prueba
}
```

Realizar pruebas: Una vez que se ha configurado el entorno de prueba y se han inyectado las dependencias necesarias, se pueden realizar pruebas utilizando los métodos de aserción de JUnit o herramientas adicionales como MockMvc.

```
@SpringBootTest
public class MyIntegrationTest {
   @Autowired
   private MyService myService;

   @MockBean
   private MyRepository myRepository;

   @Test
   public void testMyService() {
      // Configurar comportamiento simulado del repositorio
      when(myRepository.findByName("foo")).thenReturn(new MyEntity("foo", 123));

      // Llamar al servicio y verificar resultado
      MyEntity result = myService.findByName("foo");
      assertEquals(result.getName(), "foo");
      assertEquals(result.getValue(), 123);
   }
}
```

Pruebas de aceptación con Selenium

Para realizar pruebas de aceptación en Spring Boot con Selenium, se necesita configurar el entorno de prueba con una clase de prueba que cargue el contexto de Spring Boot y utilizar métodos de Selenium para interactuar con la interfaz de usuario y verificar el resultado de las pruebas.

Para realizar pruebas de aceptación en Spring Boot con Selenium, se pueden seguir los siguientes pasos:

Añadir dependencias: Para utilizar Selenium en una aplicación Spring Boot, es necesario añadir las siguientes dependencias en el archivo pom.xml:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.141.59</version>
</dependency>

<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-chrome-driver</artifactId>
  <version>3.141.59</version>
</dependency>
```

Crear una clase de prueba: Se crea una clase de prueba JUnit para realizar las pruebas de aceptación. Se utiliza la anotación @RunWith(SpringRunner.class) para cargar el contexto de Spring Boot y la anotación @SpringBootTest para indicar que se deben cargar todas las configuraciones de la aplicación.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyAcceptanceTest {
   // Aquí van los métodos de prueba
}
```

Configurar el entorno de prueba: Se utiliza la anotación @LocalServerPort para obtener el puerto aleatorio que se utiliza en la aplicación y la anotación @Before para crear el objeto WebDriver que se utilizará en las pruebas.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyAcceptanceTest {
   @LocalServerPort
   private int port;

   private WebDriver driver;

   @Before
   public void setUp() {
      driver = new ChromeDriver();
   }

   // Aquí van los métodos de prueba
}
```

Realizar pruebas: Se utilizan los métodos de Selenium para interactuar con la interfaz de usuario y verificar el resultado de las pruebas.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyAcceptanceTest {
   @LocalServerPort
   private int port;

   private WebDriver driver;

   @Before
   public void setUp() {
      driver = new ChromeDriver();
   }

   @After
   public void tearDown() {
      driver.quit();
   }

   @Test
```

```
    public void testMyPage() {
        // Abrir la página y verificar título
        driver.get("http://localhost:" + port + "/my-page");
        assertEquals("My Page", driver.getTitle());

        // Rellenar formulario y enviar
        WebElement input = driver.findElement(By.id("my-input"));
        input.sendKeys("foo");
        WebElement submit = driver.findElement(By.id("my-submit"));
        submit.click();

        // Verificar resultado
        WebElement result = driver.findElement(By.id("my-result"));
        assertEquals("foo", result.getText());
    }
}
```

Pruebas de rendimiento en Spring Boot

Las pruebas de rendimiento en Spring Boot se realizan para evaluar el comportamiento de una aplicación bajo diferentes cargas de trabajo y para identificar cuellos de botella y áreas de mejora en términos de rendimiento. A continuación, se describen los pasos generales para realizar pruebas de rendimiento en Spring Boot:

- **Identificar los casos de uso**: Identificar los casos de uso de la aplicación que se desean probar y crear los escenarios correspondientes para simular diferentes cargas de trabajo. Por ejemplo, se puede simular el acceso simultáneo a la aplicación de cientos de usuarios, realizar solicitudes de alta carga en la base de datos o enviar solicitudes de red pesadas.
- **Preparar los datos**: Preparar los datos necesarios para las pruebas, como los datos de usuario, los datos de prueba de la base de datos, etc. Asegurarse de que los datos estén limpios y sean coherentes para obtener resultados precisos y confiables.
- **Configuración del entorno**: Configurar el entorno para realizar las pruebas, que incluye la configuración del servidor de aplicaciones, la configuración de la base de datos, la configuración de la red, etc.
- **Ejecutar las pruebas**: Ejecutar las pruebas con la ayuda de herramientas de pruebas de rendimiento, como JMeter, Gatling, Apache Bench, etc. En estas pruebas, se simula la carga de trabajo para el escenario de prueba correspondiente y se miden diferentes parámetros, como el tiempo de respuesta, la tasa de errores, la carga de CPU, la memoria y el rendimiento de la base de datos.
- **Analizar los resultados**: Analizar los resultados de las pruebas para identificar áreas problemáticas en la aplicación y determinar cómo se pueden mejorar. Estos resultados también se pueden utilizar para establecer umbrales de rendimiento y comparar el rendimiento con versiones anteriores de la aplicación.
- **Optimización y ajuste**: Utilizar los resultados de las pruebas para optimizar y ajustar la aplicación para mejorar el rendimiento. Esto puede implicar ajustar la configuración del servidor, ajustar la configuración de la base de datos, optimizar el código, etc.