
DATUM

El día de ayer vimos:

- RestController
- PathVariable
- RequestParam
- Método Get
- Verbos Principales Http.

Agenda del día

- Método Post y Put
- Conexión a base de datos
- Entidades con JPA para persistir una clase Java
- Repository y Controlador del Proyecto
- Realización de consultas nativas

Método Post

RequestMapping

```
@RequestMapping(
  value = "/api/endpoint",
  method = RequestMethod.POST
)
@ResponseBody
public String verboPost() {
    return "Utilización del verbo POST";
}
```

PostMapping

```
@PostMapping(path={"/index","/", ""})
    public String primerEndpoint() {
        return "Hola Mundo ";
}
```

Método Put

RequestMapping

```
@RequestMapping(
  value = "/api/endpoint",
  method = RequestMethod.PUT
)
@ResponseBody
public String verboPut() {
    return "Utilización del verbo Put";
}
```

PutMapping

```
@PutMapping(path={"/index","/", ""})
    public String primerEndpoint() {
        return "Hola Mundo ";
}
```

Script Base de datos

```
-- Crear la base de datos
CREATE DATABASE IF NOT EXISTS Spring;
-- Seleccionar la base de datos
USE Spring;
-- Crear la tabla "Empleados"
CREATE TABLE IF NOT EXISTS Empleados (
    identificador INT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL
);
```

Conexión Base de datos

Variables Globales:

```
spring.datasource.url=jdbc:mysql://localhost:3306/SPRING
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
spring.jpa.show-sql=true
```

Dependencias

Spring Boot Starter Data JPA

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

MySQL Connector/J

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Lombok

Es una biblioteca para Java que nos ofrece bastantes funcionalidades. Para ello tiene que estar conectado a nuestro compilador, ya que su objetivo es facilitarnos el desarrollo de nuestro código evitándonos tener que escribir ciertos métodos, que van a ser repetitivos y que realmente tampoco aportan lógica al negocio.

```java
import java.util.Date;

import lombok.Data;

@Data
public class empleados{

    private Long identificador;
    private String nombre;
    private String direccion;


}
```

Entidad

Una entidad representa una tabla almacenada en una base de datos. Cada instancia de una entidad representa una fila en la tabla. Las entidades persistentes se identifican mediante la anotación @Entity.

Table

Esta anotación nos permite decidir contra que tabla de la base de datos nuestra entidad se va a mapear . En muchas situaciones la entidad no tiene el mismo nombre exacto de la base de datos y nos viene bien usar esta anotación.

Column

Nos permitirá definir aspectos muy importantes sobre las columnas de la base de datos de la base de datos como lo es el nombre. En caso de no definir esta anotación en los atributos, JPA determinara el nombre de la columna de forma automática mediante el nombre del atributo.

EntityScan

Se utiliza cuando las clases de entidades no se encuentran en el paquete principal de la aplicación ni en sus subpaquetes. En esta situación, declararíamos el paquete o la lista de paquetes en la clase de configuración principal dentro de la anotación @EntityScan.

Objeto de Acceso de Datos

Es una interfaz que proporciona métodos abstractos para acceder y manipular datos en una fuente de datos, como una base de datos. Un DAO generalmente incluye sentencias nativas SQL y, a veces, operaciones adicionales relacionadas con la persistencia de datos.

Repositorio

CrudRepository: Se trata de un repositorio genérico. Esto significa que contiene métodos aplicables a cualquier clase del dominio, pues esos métodos tienen un tipado genérico, el mismo que Repository.

Repositorio

JpaRepository: extiende de CrudRepository e incluye algunas características adicionales específicas de JPA (Java Persistence API), que es una especificación estándar de Java para la gestión de la persistencia de datos.

Query

Hay situaciones en las que necesitamos métodos muy específicos y nos puede venir bien usar la anotación de @Query que nos permite diseñar el método a medida a través de JPA.

```
@Query(value = "SELECT * FROM EMPLEADO WHERE IDENTIFICADOR = :id", nativeQuery = true)
public empleado obtenerPorId(@Param("id") Long id);
```

Param

La vinculación entre marcador y parámetro es configurable con la anotación @Param, en Spring Boot si obvias @Param cuando marcador y parámetro compartan el nombre, puede producir el siguiente error: org.springframework.dao.InvalidDataAccessApiUsageException

```
@Query(value = "SELECT * FROM EMPLEADO WHERE IDENTIFICADOR = :id", nativeQuery = true)
public empleado obtenerPorId(@Param("id") Long id);
```

Query

Al igual que utilizamos la anotación Query para realizar consultas SQL nativas, está anotación nos brinda la facilidad por medio de JPA de realizar consultas utilizando la entidad y sus atributos como valores para la consulta.

```
@Query(value="SELECT emp FROM Empleado emp WHERE emp.nombre = :nombre")
public Empleado obtenerEmpleadoPorNombre(@Param("nombre")String nombre);
```

Logger

La clase Logger de Java permite crear mensajes para el seguimiento o registro de la ejecución de una aplicación. Puede resultar de utilidad, por ejemplo, para realizar la depuración de la aplicación si se muestran los distintos puntos o estados por lo que va pasando la ejecución con los valores tomados por variables de interés.

Qualifier

Se utiliza para especificar el nombre del bean que se quiere utilizar cuando hay varios beans del mismo tipo o simplemente cuando queremos declarar particularmente el bean a utilizar.

```
@Autowired
@Qualifier("EmpleadoServicios")
EmpleadoInterfaces empleadoServicio;
```

ObjectMapper

Es una clase utilizada para leer y escribir datos JSON. Es responsable de la lectura de datos desde o hacia una Estructura de datos o Entidad, hacia un Modelo de Árbol JSON.

Características:

- Admite conceptos avanzados como Polimorfismo y Reconocimiento de objetos.
- Es muy personalizable para trabajar con diferentes estilos de contenido JSON.

Request Body

Esta anotación indica que Spring debe deserializar el cuerpo de la petición HTTP y vincularlo a un parámetro de método. El cuerpo de la solicitud se pasa a través de HttpMessageConverter para resolver el argumento del método según el tipo de contenido de la solicitud.

```
@PostMapping(path="/crearEmpleado", consumes=MediaType.APPLICATION_JSON_VALUE, produces=MediaType.APPLICATION_JSON_VALUE)
public RespuestaServicio crearEmpleado(@RequestBody EmpleadoModelo crearEmpleado) {
```

Spring Boot Starter Security

Se utiliza para incorporar características de seguridad en una aplicación Spring Boot. Spring Security es un marco de seguridad integral que se utiliza para proteger aplicaciones basadas en Spring, incluidas las aplicaciones Spring Boot.

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

PasswordEncoder

Es una interfaz de servicio en Spring Security que se utiliza para codificar contraseñas. Su función principal es codificar las contraseñas de los usuarios antes de almacenarlas en la base de datos. La codificación de contraseñas es esencial para la seguridad, ya que evita el almacenamiento de contraseñas en texto plano.

```
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

BCryptPasswordEncoder

Es una implementación concreta de la interfaz PasswordEncoder que utiliza la función de hash fuerte BCrypt. BCrypt es un algoritmo de hash de contraseñas diseñado para ser lento y resistente a ataques de fuerza bruta.

```
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

SecurityFilterChain

Se encarga de proteger las URLs de la aplicación, validar los nombres de usuario y contraseñas enviados, redirigir a formularios de inicio de sesión, entre otras tareas relacionadas con la seguridad.

```
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    http.csrf(csrf -> csrf.disable()).authorizeHttpRequests((authorize) -> {
        authorize.anyRequest().authenticated();
    }).httpBasic(Customizer.withDefaults());
    return http.build();
}
```

HttpSecurity

Se utiliza para definir reglas de seguridad que controlan qué recursos están protegidos y cómo se aplican las restricciones de acceso. Puede configurar puntos como autenticación, autorización, manejo de sesiones, entre otras opciones.

```
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    http.csrf(csrf -> csrf.disable()).authorizeHttpRequests((authorize) -> {
        authorize.anyRequest().authenticated();
    }).httpBasic(Customizer.withDefaults());
    return http.build();
}
```

UserDetailsService

Utilizando esta interfaz se puede personalizar la forma en que Spring Security obtiene detalles específicos del usuario, como su nombre de usuario, contraseña y roles. Esto es esencial para la autenticación y la autorización.

```
UserDetailsService userDetailsService() {

    UserDetails admin = User.builder().username("admin").password(passwordEncoder().encode("admin")).roles("ADMIN")
            .build();

    return new InMemoryUserDetailsManager(admin);
}
```

Swagger

Swagger permite generar automáticamente documentación para tus APIs basándose en los comentarios del código fuente. Esto facilita la comprensión de la API, los parámetros necesarios, las respuestas esperadas. La documentación es interactiva y puede probarse directamente desde la interfaz de Swagger.

DATUM

Gracias por la Atención