Ejercicio 12 – Métodos HTTP en la API REST

En la práctica anterior comenzamos a desarrollar una API REST con Spring Boot donde pusimos el foco en conceptos como la estructura Controller-Service-Repository, la configuración de la base de datos y las entidades que mapean sus tablas.

En lo relacionado a la API REST, creamos un RestController que recibía peticiones GET para hacer lecturas sobre los pilotos contenidos en la base de datos en el endpoint con URL */api/drivers*.

En esta práctica, vamos a completar el diseño de la aplicación añadiendo los métodos HTTP restantes para realizar el resto de operaciones CRUD sobre la tabla Drivers.

Parte 1: Método GET

El método GET se utiliza para obtener información de un recurso en particular. Cuando se realiza una solicitud GET a un servidor, este devuelve los datos solicitados del recurso especificado. Por ejemplo, si tenemos un servicio web que proporciona información sobre libros, una solicitud GET a */books/1* podría devolver los detalles del libro con el ID 1.

Como ya se ha comentado, en la práctica anterior utilizamos el método GET para obtener información de todos los pilotos, así que en ahora vamos a crear una nueva funcionalidad que permita consultar los datos de un piloto dado su código de piloto.

Vamos a ir paso a paso para implementar esta consulta.

Capa de Repositorio

En la interfaz del repositorio ***DriverRepository***, añade el siguiente método de consulta. Aquí, ***findByCodeIgnoreCase*** es un nombre de método especial que Spring Data JPA interpreta para generar automáticamente la consulta que queremos.

```
@Repository
public interface DriverRepository extends JpaRepository<Driver, Long> {
    //Optional<Driver> findDriverByCodeIgnoreCase(String code);
    1 usage
    Optional<Driver> findByCodeIgnoreCase(String code);
}
```

Ten en cuenta que es posible que Intellij te devuelva un error con ese nombre de método debido a que en el modelo no tenemos definida la propiedad ***@Column(name = "code")*** para el atributo code. Puedes añadir la anotación en el modelo o utilizar el nombre de método ***findDriverByCodeIgnoreCase*** en el repositorio, siendo preferible la primera opción.

El parámetro de retorno es *Optional*, ideal para trabajar con programación dinámica. ¡OJO! Podemos utilizar opcionales cuando haya una coincidencia o ninguna, pero no si hubiera posibles varias coincidencias, así que asegúrate de que trabajas en una columna con la propiedad *Unique*.

Capa de Servicio

Primero, añade en la interfaz del servicio ***DriverService*** el método abstracto que pueda ser implementado a posteriori y utilizado por el controlador.

```
public interface DriverService {
    1 usage  1 implementation
    List<Driver> getAllDrivers();
    1 usage  1 implementation
    Optional<Driver> getDriverByCode(String code);
}
```

A continuación, impleméntala en la clase ***DriverServiceImpl*** utilizando el método anterior del repositorio.

```
@Override
public Optional<Driver> getDriverByCode(String code) {
    return repository.findByCodeIgnoreCase(code);
}
```

Capa de Controlador

Por último, en el controlador REST ***DriverRestController***, añadimos el endpoint ***/api/drivers/{code}***, que llama al método del servicio para obtener el piloto con el código indicado.

```
/*
GET http://localhost:8080/api/drivers/alo
 */
@GetMapping("/drivers/{code}")
public ResponseEntity<Driver> getByCode(@PathVariable String code) {
    return this.driverService.getDriverByCode(code) Optional<Driver>
            .map(ResponseEntity::ok) Optional<ResponseEntity<...>>
            .orElse(ResponseEntity.notFound().build());
}
```

Respecto al código anterior, varias aclaraciones:

1. ***@PathVariable*** es una anotación de Spring que se utiliza para extraer valores de la URL y asignarlos a parámetros de método en el controlador. En el código, ***@PathVariable String code*** indica que el valor de la variable code se tomará de la porción de la URL que coincide con /drivers/code/{code}. Por ejemplo, si la URL es /drivers/code/alo, entonces code tomará el valor "alo".

2. El método getDriverByCode(code) devuelve un ***Optional***, que nos permite manejar de manera más elegante los casos donde un valor puede ser nulo. Con el método ***.map(ResponseEntity::ok)***, si hay un valor presente, se mapea a un ResponseEntity con estado ***OK (200)*** y el valor del controlador como el cuerpo de la respuesta. En caso de que el Optional esté vacío, con ***.orElse(ResponseEntity.notFound().build())*** se construye un ResponseEntity con estado ***NOT_FOUND (404)***.

Parte 2: Métodos POST, PUT y DELETE

Además del método GET, son fundamentales para el diseño de APIs RESTful los siguientes métodos:

- ***POST***: Se utiliza para enviar datos a un servidor y **crear un nuevo recurso**. Los datos a enviar se serializan y se incluyen en el cuerpo de la solicitud. Siguiendo el ejemplo anterior, podríamos usar POST para añadir un nuevo libro a la colección, enviando los detalles del libro (título, autor, fecha de publicación, etc.) en el cuerpo de la solicitud a "*/books*".

- ***PUT***: Se utiliza para **actualizar un recurso existente**. Al igual que POST, los datos a enviar se incluyen en el cuerpo de la solicitud. Por ejemplo, podríamos utilizar PUT para actualizar los detalles del libro con el ID 1, enviando los nuevos detalles en el cuerpo de la solicitud a "*/books/1*".

- ***DELETE***: Se utiliza para **eliminar un recurso**. Para ello, especificamos el recurso que deseamos eliminar. Por ejemplo, podríamos utilizar DELETE para eliminar el libro con el ID 1 haciendo una solicitud DELETE a "*/books/1*".

A continuación, tienes la parte del controlador a añadir para crear los endpoints:

```java
/*
POST http://localhost:8080/api/drivers/
*/
@PostMapping("/drivers")
public ResponseEntity<Driver> create(@RequestBody Driver driver) {
    if (driver.getDriverId() != null)
        return ResponseEntity.badRequest().build();

    this.driverService.saveDriver(driver);
    return ResponseEntity.ok(driver);
}

/*
PUT http://localhost:8080/api/drivers/
*/
@PutMapping("/drivers")
public ResponseEntity<Driver> update(@RequestBody Driver driver) {
    this.driverService.saveDriver(driver);
    return ResponseEntity.ok(driver);
}

/*
DELETE http://localhost:8080/api/drivers/alo
*/
@DeleteMapping("/drivers/{code}")
public ResponseEntity<Driver> deleteByCode(@PathVariable String code) {
    this.driverService.deleteDriverByCode(code);
    return ResponseEntity.noContent().build();
}
```

Completa el resto de capas y prueba el funcionamiento de la aplicación con Postman o RESTer.