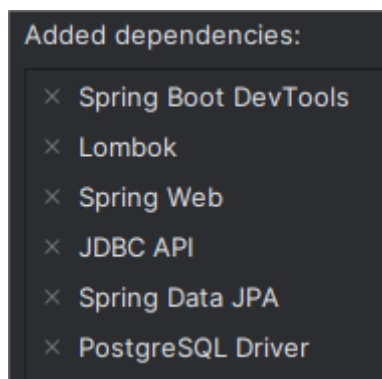


Ejercicio 11 – Mi primera API REST

En esta práctica vamos a crear una API REST con Spring Boot, facilitándonos las configuraciones y las dependencias de conexión con la base de datos.

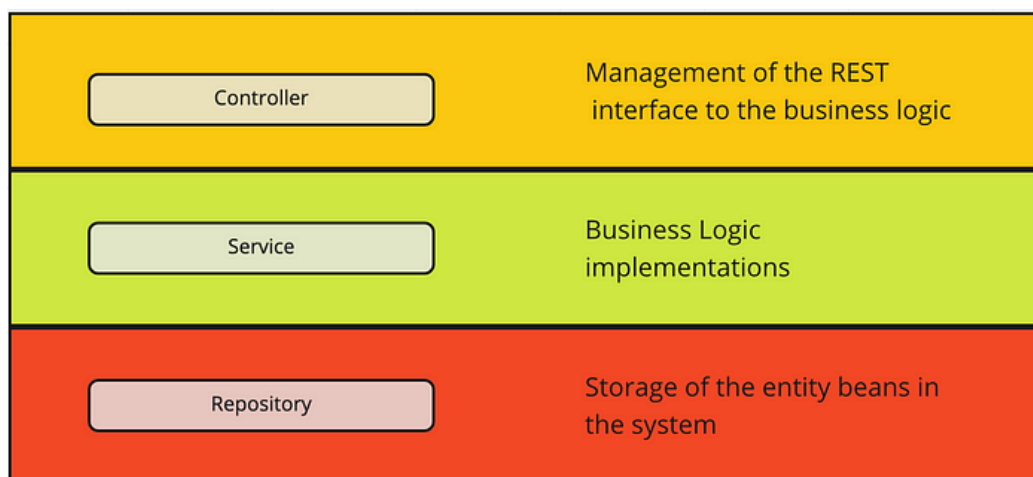
Parte 1: Crear la aplicación

Como en el proyecto de la práctica anterior, iniciaremos nuestro proyecto mediante Spring Initializr. Dentro de las dependencias selecciona las siguientes:



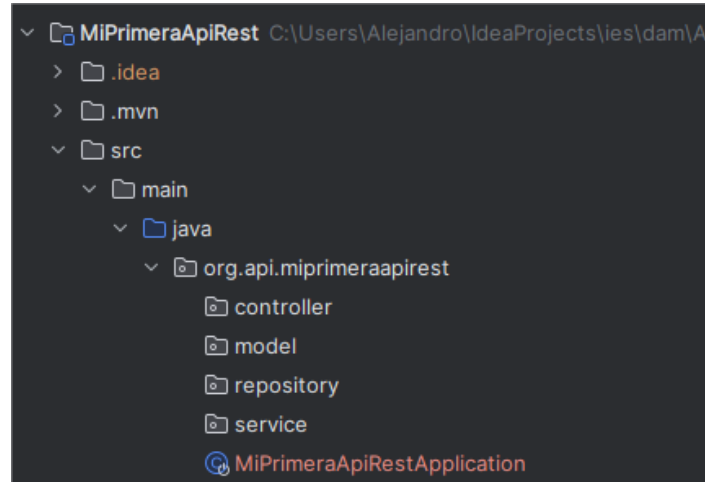
Parte 2: Estructura de paquetes

En la parte teórica vimos que muchas aplicaciones de Spring Boot utilizan el patrón Controller-Service-Repository para separar las responsabilidades.



- La capa **Controller**, la capa superior de la imagen, es la única responsable de exponer la funcionalidad para que pueda ser consumida por entidades externas, pudiendo incluir un componente de interfaz de usuario.
- La capa **Repository**, la capa inferior de la imagen, es responsable de almacenar y recuperar datos desde las distintas fuentes de datos.
- La capa **Service** es donde debe ir toda la lógica de negocio. Si la lógica de negocio requiere recuperar/guardar datos, se conecta a un repositorio. Si alguien quiere acceder a esta lógica de negocio, pasa por un controlador para llegar allí.

En base a ello, vamos a crear un paquete para cada una de estas tres capas. Además, agregaremos un cuarto paquete denominado **model**, donde definiremos nuestras entidades con sus atributos, sus claves y cómo se relacionan entre ellas.



Parte 3: La base de datos

Para este proyecto vamos a reutilizar la base de datos PostgreSQL que instalaste en AWS sobre la temporada 2006 de F1. En caso de haberla borrado, vuelve a revisar el ejercicio 7 donde tendrás el fichero SQL y las instrucciones para volcar el script en una instancia RDS en AWS.

En Spring Boot, la conexión a la base de datos y otras configuraciones relacionadas con JPA se definen en el archivo **application.properties** o **application.yml** que se encuentra en el directorio **src/main/resources**.

A continuación tenemos un ejemplo de cómo podríamos configurar la conexión a nuestra base de datos:

```
1 spring.datasource.url=jdbc:postgresql://aadd-f12006.cz60880m2nb2.us-east-1.rds.amazonaws.com/f12006
2 spring.datasource.username=postgres
3 spring.datasource.password=qwerty1234
4
5 spring.jpa.show-sql=true
```

En este ejemplo, **spring.datasource.url** es la URL de la base de datos, **spring.datasource.username** y **spring.datasource.password** son el nombre de usuario y la contraseña para la base de datos.

La propiedad **spring.jpa.show-sql** controla si se deben mostrar las consultas SQL en la consola.

Otra propiedad muy utilizada es **spring.jpa.hibernate.ddl-auto**, que controla el comportamiento de generación automática de esquemas. Aquí están las opciones que puedes usar:

- **none**: No hace nada.
- **validate**: Verifica que las tablas correspondientes a las entidades existan, pero no las crea ni las modifica.
- **update**: Crea las tablas que no existen y actualiza las que necesitan ser actualizadas, pero no borra ninguna tabla existente.

- **create**: Crea las tablas al inicio de la aplicación, pero no las borra cuando se cierra la aplicación.
- **create-drop**: Crea las tablas al inicio de la aplicación y las borra cuando se cierra la aplicación.

Parte 4: Entidades, modelos y esquemas

En Spring Data JPA, una entidad es una clase Java que se mapea a una tabla en una base de datos relacional. De esta manera podemos definir nuestros esquemas de la base de datos mediante código. Para definir una entidad en JPA, se deben seguir los siguientes pasos:

1. Anotar la clase Java con la anotación **@Entity**: Esto indica que la clase Java es una entidad que se debe mapear a una tabla en la base de datos.
2. Especificar el nombre de la tabla en la base de datos: Puedes especificar el nombre de la tabla usando la anotación **@Table**.
3. Especificar el identificador de la entidad: Cada entidad en JPA debe tener un identificador único que se utiliza para acceder y manipular la entidad. El identificador se puede especificar usando la anotación **@Id**. Además, es posible que necesites especificar el tipo de identificador usando otras anotaciones, como **@GeneratedValue** si quieres que el identificador se genere automáticamente.
4. Definir los atributos de la entidad: Los atributos corresponden a las columnas en la tabla de la base de datos. Puedes especificar propiedades de la columna con la anotación **@Column**.

Más adelante veremos que también podemos definir restricciones de integridad para columnas siguiendo los validadores existentes en JPA y de la especificación de Bean Validation (más info [aquí](#)).

Cae la clase **Driver** en el paquete **model**. Esta clase mapeará los atributos como campos en la base de datos.

```
@Entity
@Data
@Table(name = "drivers")
public class Driver {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "driverid")
    private Long driverId;

    @Column(unique = true)
    private String code;
    private String forename;
    private String surname;
    @JsonProperty("dateOfBirth")
    private LocalDate dob;
    private String nationality;
    private String url;
}
```

Parte 5: Repositorio

En Spring Data JPA, un repositorio (anotado con **@Repository**) es una interfaz que define una colección de métodos para acceder y manipular datos en una base de datos relacional. Un repositorio permite abstraer la capa de acceso a datos y facilita la implementación de operaciones CRUD (crear, leer, actualizar, eliminar) en la aplicación. De hecho, ya tiene implementados los métodos CRUD básicos, por lo que no es necesario escribir código adicional para interactuar con la base de datos.

Sin embargo, si queremos métodos más específicos podemos hacer uso de la anotación **@Query** que permite definir consultas personalizadas y personalizar las consultas generadas automáticamente.

Existen diferentes tipos de repositorios en Spring Data JPA, algunos de los más comunes son:

- **JpaRepository**: es una interfaz que extiende la interfaz CrudRepository y agrega funcionalidades específicas para JPA. Proporciona operaciones como guardar, eliminar, actualizar y buscar, además de soportar paginación, queries y ordenamiento de los resultados.
- **PagingAndSortingRepository**: es una interfaz que extiende la interfaz CrudRepository y agrega soporte para paginación y ordenamiento de los resultados.
- **CrudRepository**: es la interfaz más básica y proporciona las operaciones CRUD básicas, como guardar, eliminar, actualizar y buscar.

Crea la interfaz **DriverRepository** en el paquete **repository** con el siguiente contenido:

```
@Repository
public interface DriverRepository extends JpaRepository<Driver, Long> {
}
```

Como puedes comprobar, la interfaz está vacía porque las consultas se generan automáticamente con la interfaz CrudRepository. Puedes consultar información sobre dichos métodos en [este enlace](#).

La generación automática de consultas hace que tampoco necesitemos una clase que implemente dichas consultas. En caso de generar nuestras propias consultas en la interfaz deberíamos crear una clase **DriverRepositoryImpl** que las implemente.

Parte 6: Servicio

El servicio es el intermediario entre el repositorio y el controlador, que gestionará las peticiones de la API en el siguiente apartado.

Crea una interfaz **DriverService** en el paquete **service** con el siguiente contenido:

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

Esta interfaz define un único método, el cuál será implementado por la clase **DriverServiceImpl**, el cuál tendrá la anotación **@Service** para indicar que es un servicio y también la anotación **@Autowired** para inyectar el repositorio y hacer uso de él:

```
@Service
public class DriverServiceImpl implements DriverService {
    2 usages
    private final DriverRepository repository;

    @Autowired
    public DriverServiceImpl(DriverRepository repository) {
        this.repository = repository;
    }

    1 usage
    @Override
    public List<Driver> getAllDrivers() {
        return repository.findAll();
    }
}
```

La inyección de dependencia de Spring se puede lograr a través del constructor, el método Setter y el dominio de entidad. El equipo de Spring generalmente aconseja la inyección por constructor, como se ha hecho en el código.

Parte 7: Controlador

Por último, en el paquete **controller** tendremos la clase **DriverRestController**, un controlador de tipo **RestController** que se encargará de recibir las peticiones y devolver las respuestas en formato por defecto, en este caso JSON. Para ello, utilizaremos la anotación **@RestController**.

Si llegados a este punto no conoces que es REST o no has trabajado nunca con esta tecnología, te recomiendo encarecidamente leer [el siguiente artículo](#) antes de continuar.

La definición de clase del controlador tiene también una anotación **@RequestMapping**, indicando la URL base. En nuestro caso sería **http://localhost:8080/api/**

Las peticiones que vamos a recibir seguirán los verbos HTTP que conocemos: tipo **GET (GetMapping)**, **POST (PostMapping)**, **PUT (PutMapping)**, **PATCH (PatchMapping)** y/o **DELETE (DeleteMapping)**. De esta manera podremos hacer las peticiones CRUD que necesitemos.

Para devolver las respuestas vamos a usar la clase **ResponseEntity**. Esta clase nos permite devolver el código de estado HTTP de la respuesta, así como el cuerpo de la misma. Podemos devolver respuestas concretas con los **HttpStatus** que necesitemos: **OK**, **CREATED**, **BAD_REQUEST**, **NOT_FOUND**, etc.

En nuestro código, empezaremos simplemente por ofrecer una lista de **Driver** en **/api/drivers** con los elementos mencionados anteriormente:

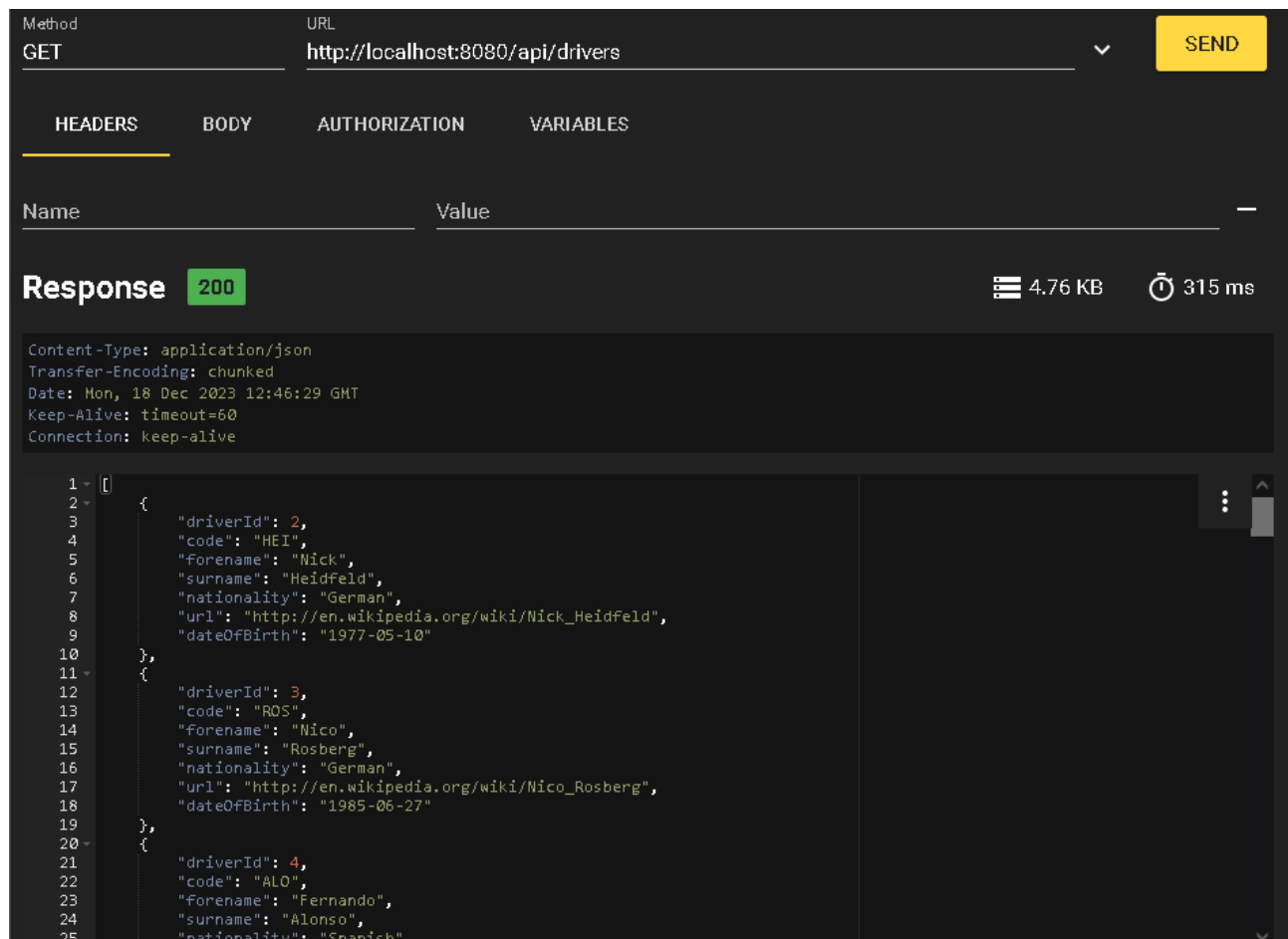
```
@RestController
@RequestMapping("/api")
public class DriverRestController {
    2 usages
    private final DriverService driverService;

    @Autowired
    public DriverRestController(DriverService service) {
        this.driverService = service;
    }

    @GetMapping("/drivers")
    public ResponseEntity<List<Driver>> getAll() {
        return ResponseEntity.ok(driverService.getAllDrivers());
    }
}
```

Parte 8: Funcionamiento de la API

Para probar con un cliente nuestro servicio podemos usar **Postman** que es una herramienta de colaboración para el desarrollo de APIs. Otra opción, y válida en Lliurex, es instalar la extensión **RESTER** en el navegador Firefox desde [aquí](#). Esta extensión también está disponible en Google Chrome si así lo requieres.



The screenshot shows the Postman interface. The top bar indicates the method is GET and the URL is http://localhost:8080/api/drivers. The 'SEND' button is visible. Below the URL bar, the 'HEADERS' tab is selected, showing no headers. The 'BODY' tab is also visible. The 'Response' section shows a status code of 200, a size of 4.76 KB, and a time of 315 ms. The response body is a JSON array of three driver objects:

```
1 - [
2   {
3     "driverId": 2,
4     "code": "HEI",
5     "forename": "Nick",
6     "surname": "Heidfeld",
7     "nationality": "German",
8     "url": "http://en.wikipedia.org/wiki/Nick_Heidfeld",
9     "dateOfBirth": "1977-05-10"
10  },
11  {
12    "driverId": 3,
13    "code": "ROS",
14    "forename": "Nico",
15    "surname": "Rosberg",
16    "nationality": "German",
17    "url": "http://en.wikipedia.org/wiki/Nico_Rosberg",
18    "dateOfBirth": "1985-06-27"
19  },
20  {
21    "driverId": 4,
22    "code": "ALO",
23    "forename": "Fernando",
24    "surname": "Alonso",
25    "nationality": "Spanish",
```