SPRING MYSQL DOCKER

Proyecto API REST de clientes utilizando Spring Boot con MySQL y Docker

Este proyecto crea una API REST que devuelve información de clientes utilizando Spring y una base de datos MySQL que se ejecuta desde un contenedor Docker.

Iniciar proyecto con Spring Initializr

Para iniciar el proyecto, vamos a la página start.spring.io para crear nuestro proyecto de Spring Boot de manera sencilla.

Configuración del proyecto inicial

En el apartado que aparece dentro de la página configuramos el proyecto:

- **Project**: Maven Project
- **Language**: Java
- **Spring Boot**: La versión estable más reciente. *A 26 de julio de 2025 es la 3.5.4*
- **Project Metadata**:
  - **Group**: *com.academy*
  - **Artifact**: *rest*
  - **Name**: *rest*
  - **Description**: *Descripción del proyecto*
  - **Package name**: *com.academy.rest*
- **Packaging**: Jar
- **Java**: 17. *Utilizar la versión de nuestra elección para el proyecto.*

Añadiendo dependencias

Agregamos las dependencias necesarias para las funcionalidades que se implementarán.

- **Spring Web**: Para construir aplicaciones web y APIs REST.
- **Spring Data JPA**: Para acceso a datos y persistencia.
- **MySQL Driver**: Para la conexión con la base de datos MySQL.

Agregadas todas las dependencias presionamos el botón **Generate** para descargar el zip del proyecto.

Creación de contenedor de la base de datos con Docker

Creación de contenedor MySQL

Para crear el contenedor de la base de datos ejecutamos el siguiente comando:

```
docker run --name mysql-db -p 3306:3306 -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=clientes_db -d mysql:8.0
```

Con este comando configuramos lo siguiente:

- **`--name mysql-db`**: Asigna el nombre `mysql-db` al contenedor.
- **`-p 3306:3306`**: Mapea el puerto 3306 del host al puerto 3306 del contenedor.
- **`-e MYSQL_ROOT_PASSWORD=root`**: Establece la contraseña del usuario `root` de MySQL.
- **`-e MYSQL_DATABASE=clientes_db`**: Crea una base de datos llamada `clientes_db`.
- **`-d`**: Ejecuta el contenedor en modo detached (en segundo plano).
- **`mysql:8.0`**: Especifica la imagen de MySQL a utilizar. *En caso de que no exista la imagen, la descargará.*

Creación y configuración de base de datos en MySQL

Una vez creado el contenedor, podemos usar las credenciales para conectarnos a la base de datos a través de una herramienta de gestión. En mi caso, utilicé DBeaver.

Creación de usuario y base de datos

El primer paso será crear un usuario llamado `xideral`, una base de datos llamada `academy_db` y otorgarle todos los privilegios sobre esa base de datos al usuario. Esto se logra con el siguiente script:

```sql
CREATE USER 'xideral'@'%' IDENTIFIED BY 'xideral';

CREATE DATABASE IF NOT EXISTS academy_db;

GRANT ALL PRIVILEGES ON academy_db.* TO 'xideral'@'%';

FLUSH PRIVILEGES;
```

Explicación de script

- `GRANT ALL PRIVILEGES ON academy_db.* TO 'xideral'@'%';`: Crea un nuevo usuario llamado `xideral` que puede conectarse desde cualquier host (%), con la contraseña `xideral`.
- `CREATE DATABASE IF NOT EXISTS academy_db;`: Crea la base de datos `academy_db` solo si no existe previamente.
- `GRANT ALL PRIVILEGES ON academy_db.* TO 'xideral'@'%';`: Asigna todos los privilegios (`SELECT`, `INSERT`, `UPDATE`, etc.) sobre la base de datos `academy_db` al usuario `xideral` desde cualquier host.
- `FLUSH PRIVILEGES;`: Recarga los privilegios en el servidor para aplicar los cambios realizados con `GRANT`.

🐬 Creación de tabla

El siguiente paso es crear nuestra tabla `Cliente`, que incluye *id* (clave primaria), *nombre*, *email* (único), *teléfono* y *fecha de registro* con timestamp automático. Para ello utilizamos el siguiente script:

```sql
CREATE TABLE Cliente (
      id INT AUTO_INCREMENT PRIMARY KEY,
      nombre VARCHAR(100) NOT NULL,
      email VARCHAR(150) UNIQUE NOT NULL,
      telefono VARCHAR(20),
      fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  );
```

Inserción de datos

Finalmente insertamos 10 registros en la tabla Cliente con este script para probar nuestra aplicación cuando sea necesario.

```sql
INSERT INTO Cliente (nombre, email, telefono) VALUES
  ('Juan Pérez', 'juan.perez@email.com', '555-0101'),
  ('María García', 'maria.garcia@email.com', '555-0102'),
  ('Carlos López', 'carlos.lopez@email.com', '555-0103'),
  ('Ana Martínez', 'ana.martinez@email.com', '555-0104'),
  ('Pedro Rodríguez', 'pedro.rodriguez@email.com', '555-0105'),
  ('Laura Fernández', 'laura.fernandez@email.com', '555-0106'),
  ('Miguel Sánchez', 'miguel.sanchez@email.com', '555-0107'),
  ('Sofia Ruiz', 'sofia.ruiz@email.com', '555-0108'),
  ('Diego Morales', 'diego.morales@email.com', '555-0109'),
  ('Carmen Jiménez', 'carmen.jimenez@email.com', '555-0110');
```

⚙️ Configuración de propiedades del proyecto

Extraemos el zip previamente descargado de Spring Initializr y abrimos con el IDE de nuestra preferencia.

🔧 Configuración de conexión a la base de datos

Abrimos el archivo `application.properties` ubicado en la ruta `src/main/resources` y añadimos las siguientes propiedades.

```
spring.application.name=rest

# MySQL Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/academy_db
spring.datasource.username=root
spring.datasource.password=xideral2025
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA/Hibernate Configuration
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardIm

# Static resources configuration
spring.web.resources.static-locations=classpath:/static/
spring.mvc.static-path-pattern=/**
```

💡 Explicación de propiedades:

`spring.application.name=rest`: Define el nombre la aplicación Spring.

- **Configuración de base de datos:**
  - `spring.datasource.url`: Define la URL de conexión a la base de datos MySQL `academy_db` alojada en `localhost` y el puerto `3306`.
  - `spring.datasource.username`: Usuario para conectarse a la base de datos (`root`).
  - `spring.datasource.password`: Contraseña del usuario para la conexión (`xideral2025`).
  - `spring.datasource.driver-class-name`: Especifica el driver de MySQL necesario para la conexión (`com.mysql.cj.jdbc.Driver`).

- **Configuración de JPA y Hibernate:**
  - `spring.jpa.hibernate.ddl-none`: No realiza ningún cambio en el esquema de la base de datos automáticamente al iniciar. Ideal para producción.
  - `spring.jpa.show-sql`: Muestra las consultas SQL en consola para facilitar la depuración.
  - `spring.jpa.properties.hibernate.dialect`: Define el dialecto específico para generar SQL compatible con MySQL.
  - `spring.jpa.hibernate.naming.physical-strategy`: Usa nombres de tablas y columnas tal cual como están en las entidades, sin alteraciones.

- **Configuración de recursos estáticos para el front-end**
  - `spring.web.resources.static-locations`: Ubicación de archivos estáticos como HTML, CSS y JS dentro del proyecto (`classpath:/static/`).
  - `spring.mvc.static-path-pattern`: Patrón que define cómo se acceden los recursos estáticos desde el navegador (`/**`).

Una vez terminada la configuración, nuestra aplicación debería de conectarse sin problema a la base de datos.

🌐 Creación de Entidad

Primero, creamos una clase que represente los datos que queremos guardar. En nuestro caso, queremos guardar clientes. Para ello creamos la clase Cliente. Para organizar el proyecto, la inicializaremos dentro de un paquete llamado entity

A esta clase añadiremos la anotación `@Entity` para indicar que representa una entidad persistente y que está mapeada a una tabla en la BD, cuando Spring la detecta, configurá automáticamente un ORM (en este caso Hibernate) para gestionar la persistencia. También se utiliza la anotación `@Table` la cual se utiliza para personalizar el mapeo entre una clase entidad y su tabla en la base de datos.

A la anotación `@Table` esta le pasamos el atributo `name = "Cliente"` porque así se llama nuestra tabla en la base de datos.

```
package com.academy.rest.entity;

@Entity
@Table(name = "Cliente")
public class Cliente {}
```

Dentro de la clase, declaramos los atributos necesarios para mapear la clase, estos deben ser los mismos que se encuentran presentes en la tabla Cliente de la base de datos (id, nombre, email, telefono, fechaRegistro).

Agregamos las anotaciones `@Id` el cual indica que ese campo es la clave primaria (*toda anotación `@Entity` debe tener una clave primaria*), y `@GeneratedValue(strategy = GenerationType.IDENTITY)` en la cual se le delega al motor de base de datos la generación automática

de ese valor. Toda

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

En los demás campos agregamos la anotación `@Column` que se usa para personalizar el mapeo de los atributos:

```
@Column(name = "nombre", nullable = false, length = 100)
private String nombre;

@Column(name = "email", nullable = false, length = 150, unique = true)
private String email;

@Column(name = "telefono", length = 20)
private String telefono;

@Column(name = "fecha_registro", columnDefinition = "TIMESTAMP DEFAULT CURRENT_TIMESTAMP")
private LocalDateTime fechaRegistro;
```

En los atributos, `name` indica el nombre, `nullable` si el campo puede ser nulo o no, `length` el tamaño del campo y `columnDefinition` permite especificar directamente la definición SQL que se usará al generar la columna en la base de datos.

- `nombre` debe tener un máximo de 100 caracteres y no puede ser nulo;
- `email` tiene hasta 150 caracteres, es obligatorio y debe ser único en la tabla;
- `telefono` es opcional y limitado a 20 caracteres;
- `fechaRegistro` se define como un campo de tipo `TIMESTAMP` con valor por defecto asignado al momento de la inserción, gracias a columDefinition.

Posteriormente agregamos los constructores. En este caso necesitamos dos:

- Hibernate necesita un **constructor vacío** para poder funcionar correctamente.

```
public Cliente() {}
```

- Creamos también un constructor con **todos los parámetros** el cuál nos permitirá crear instancias del objeto con valores específicos.

```
public Cliente(String nombre, String email, String telefono) {
    this.nombre = nombre;
    this.email = email;
    this.telefono = telefono;
    this.fechaRegistro = LocalDateTime.now();
}
```

Es importante destacar que el constructor no necesita el parámetro `id` ya que este se le asigna automáticamente debido a la implementación de JPA y el encargado de generarlo será el motor de base de datos.

Finalmente, insertamos `getters` y `setters` para poder acceder a los atributos de la clase ya que son privados.

```
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public LocalDateTime getFechaRegistro() {
        return fechaRegistro;
    }

    public void setFechaRegistro(LocalDateTime fechaRegistro) {
        this.fechaRegistro = fechaRegistro;
    }
```

📦 Creación de Repositorio

El siguiente paso es crear el repositorio, esta interfaz es responsable de la capa de acceso a datos para la entidad Cliente.

```
@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long> {

    Optional<Cliente> findByEmail(String email);

    List<Cliente> findByNombreContainingIgnoreCase(String nombre);

    @Query("SELECT c FROM Cliente c WHERE c.telefono = :telefono")
    Optional<Cliente> findByTelefono(@Param("telefono") String telefono);

    @Query("SELECT c FROM Cliente c ORDER BY c.fechaRegistro DESC")
    List<Cliente> findAllOrderByFechaRegistroDesc();

    boolean existsByEmail(String email);
}
```

A esta interfaz añadimos la anotación `@Repository` la cual indica a Spring que es un componente de la capa de persistencia, la registra como un bean en el contenedor de inversión de control (IoC) y proporciona manejo de excepciones de persistencia.

Al extender `JpaRepository`, `ClienteRepository` hereda automáticamente métodos para operaciones CRUD (Crear, Leer, Actualizar, Borrar) sin necesidad de escribir código de implementación.

Los atributos que recibe `JpaRepository` son:

- **Cliente**: Es la entidad que gestionará este repositorio.
- **Long**: Es el tipo de dato de la clave primaria de la entidad Cliente.

Dentro de la interfaz se declaran ciertos métodos sin una implementación debido a que Spring Data JPA utiliza *Query Methods*, con los cuales analiza el nombre del método (que debe seguir una convención de nombres especifica) y genera la query especificada en tiempo de ejecución.

Algunos de estos métodos se anotan con `@Query`, en la cual se puede especificar la consulta SQL a utilizar por JPA para buscar la información en la base de datos.

⚒️ Capa de Servicio

La capa de servicio contiene la lógica de negocio de la aplicación. Separa los detalles de la gestión de datos (repositorio) de la lógica de la aplicación que los consume (controlador).

💡 Creación de interfaz

Primero creamos un paquete service y dentro la interfaz `IClienteService` la cual es un contrato para definir nuestro servicio. Se pone una "I" antes del nombre como buena práctica que identifica que se trata de una interfaz.

```
public interface IClienteService {

    List<Cliente> getAllClientes();

    Optional<Cliente> getClienteById(Long id);

    Cliente createCliente(Cliente cliente);

    Cliente updateCliente(Long id, Cliente clienteDetails);

    boolean deleteCliente(Long id);

    Optional<Cliente> getClienteByEmail(String email);

    List<Cliente> getClientesByNombre(String nombre);

    Optional<Cliente> getClienteByTelefono(String telefono);

    List<Cliente> getAllClientesOrderByFechaRegistro();
}
```

Dentro de nuestra interfaz definimos los métodos que podrá realizar nuestro servicio en su implementación.

💡 Creación de implementación de servicio

Creamos ahora la clase de implementación dentro del paquete service. Esta clase implementa la lógica de los métodos definidos en la interfaz IClienteService.

```java
@Service
public class ClienteServiceImpl implements IClienteService {

    @Autowired
    private ClienteRepository clienteRepository;

    public List<Cliente> getAllClientes() {
        return clienteRepository.findAll();
    }

    public Optional<Cliente> getClienteById(Long id) {
        return clienteRepository.findById(id);
    }

    public Cliente createCliente(Cliente cliente) {
        if (clienteRepository.existsByEmail(cliente.getEmail())) {
            throw new RuntimeException("Ya existe un cliente con el email: " + cliente.getEmail());
        }
        cliente.setFechaRegistro(LocalDateTime.now());
        return clienteRepository.save(cliente);
    }

    public Cliente updateCliente(Long id, Cliente clienteDetails) {
        Optional<Cliente> clienteOptional = clienteRepository.findById(id);
        if (clienteOptional.isPresent()) {
            Cliente cliente = clienteOptional.get();
            if (!cliente.getEmail().equals(clienteDetails.getEmail()) &&
                clienteRepository.existsByEmail(clienteDetails.getEmail())) {
                throw new RuntimeException("Ya existe un cliente con el email: " + clienteDetails.getEmail());
            }

            cliente.setNombre(clienteDetails.getNombre());
            cliente.setEmail(clienteDetails.getEmail());
            cliente.setTelefono(clienteDetails.getTelefono());

            return clienteRepository.save(cliente);
        } else {
            throw new RuntimeException("Cliente no encontrado con ID: " + id);
        }
    }

    public boolean deleteCliente(Long id) {
        if (clienteRepository.existsById(id)) {
            clienteRepository.deleteById(id);
            return true;
        } else {
            throw new RuntimeException("Cliente no encontrado con ID: " + id);
        }
    }

    public Optional<Cliente> getClienteByEmail(String email) {
        return clienteRepository.findByEmail(email);
    }

    public List<Cliente> getClientesByNombre(String nombre) {
        return clienteRepository.findByNombreContainingIgnoreCase(nombre);
    }

    public Optional<Cliente> getClienteByTelefono(String telefono) {
        return clienteRepository.findByTelefono(telefono);
    }

    public List<Cliente> getAllClientesOrderByFechaRegistro() {
        return clienteRepository.findAllOrderByFechaRegistroDesc();
    }
}
```

A la clase se le anota con `@Service` para indicarle a Spring que es un componente de la capa de servicio y así cree un bean en el contenedor de inversión de control (IoC).

Los métodos de esta clase son todos públicos ya que cuando una clase implementa una interfaz, todos los métodos de esa interfaz deben ser declarados como `public` en la clase de implementación.

Cada método en la clase implementa la lógica definida en la interfaz IClienteService. En este caso, la lógica es bastante simple y delega la llamada directamente al clienteRepository quien a su vez la delega a JPA a través de JpaRepository.

El código siguiente es muy importante ya que es un ejemplo de Inyección de Dependencias, un principio fundamental de Spring.

Se utiliza la anotación `@Autowired` para que el framework sea el encargado de realizar la inyección.

```
@Autowired
private ClienteRepository clienteRepository;
```

En lugar de que `ClienteServiceImpl` cree una instancia de `ClienteRepository`, Spring se encarga de crearla e "*inyectarla*" en este campo. Esto desacopla los componentes, haciendo que el código sea más modular, fácil de probar y mantener.

Controlador

La última parte de la creación de nuestro proyecto es la creación del controlador. Esta clase expone la funcionalidad del servicio de clientes como una API REST, manejando las solicitudes HTTP.

En palabras simples, el controlador es el que recibe las peticiones de los usuarios y les responde.

```
@RestController
@RequestMapping("/api")
public class ClienteController {
...
```

La clase se anota con `@RestController` que una anotación de conveniencia que combina `@Controller` y `@ResponseBody`.

- `@Controller`: Indica que la clase es un controlador web.
- `@ResponseBody`: Indica que el valor de retorno de los métodos debe ser serializado (generalmente a JSON) y escrito directamente en el cuerpo de la respuesta HTTP.

También se anota con `@RequestMapping("/ruta")`, esta define un prefijo de ruta para todos los endpoints en este controlador. Todas las rutas definidas dentro de esta clase comenzarán con el parámetro proporcionado a la anotación, en este caso será `/api`.

Posteriormente se utiliza nuevamente la anotación `@Autowired` para inyectar nuestro servicio al controlador, y así poder utilizarlo para realizar las operaciones de nuestra aplicación y trabajar sobre la base de datos.

```
@Autowired
private IClienteService clienteService;
```

De esta manera el controlador no se preocupa por cómo se crea o implementa IClienteService, solo lo usa.

Cada uno de los métodos del controlador emplea diferentes anotaciones dependiendo del método HTTP que las podrá invocar.

- `@GetMapping("/ruta")`: Mapea las solicitudes HTTP GET a /api/clientes hacia el método.
- `@PostMapping("/ruta")`: Mapea las solicitudes HTTP POST a /api/clientes hacia el método. Se usa para crear un nuevo recurso.
- `@PutMapping("/ruta")`: Mapea solicitudes HTTP PUT hacia el método. Se usa para actualizar un recurso existente por completo.
- `@DeleteMapping("/ruta")`: Mapea solicitudes HTTP DELETE hacia el método.Se usa para eliminar un recurso.

En el controlador todos los métodos son de tipo `ResponseEntity`, la cual es una clase que representa la respuesta HTTP completa. Da control total sobre todo lo que se envía al cliente, incluyendo:

1. El Código de Estado HTTP: Por ejemplo, `200 OK`, `404 Not Found`, `201 Created`.
2. Las Cabeceras HTTP: Por ejemplo, Content-Type, Location.
3. El Cuerpo de la Respuesta (Body): Los datos que quieres enviar de vuelta, por ejemplo, un objeto Cliente o una lista de Cliente.

```
@GetMapping
public ResponseEntity<List<Cliente>> getAllClientes() {
    List<Cliente> clientes = clienteService.getAllClientes();
    return ResponseEntity.ok(clientes);
}

@GetMapping("/{id}")
public ResponseEntity<Cliente> getClienteById(@PathVariable Long id) {
    Optional<Cliente> cliente = clienteService.getClienteById(id);
    if (cliente.isPresent()) {
        return ResponseEntity.ok(cliente.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

En el método `getAllClientes` estamos pasando un parámtero utilizando `@PathVariable`: esta anotación extrae el valor de la variable de la ruta (`{id}`) y lo convierte en un `Long`.

Después en `getClienteById` retornamos métodos de ResponseEntity condicionalmente, si existe un cliente retornaremos `ResponseEntity.ok()` que devolverá un código HTTP 200, caso contrario retornaremos `ResponseEntity.notFound()` que devuelve un código HTTP 404.

```
    @PostMapping
    public ResponseEntity<?> createCliente(@RequestBody Cliente cliente) {
        try {
            Cliente nuevoCliente = clienteService.createCliente(cliente);
            return ResponseEntity.status(HttpStatus.CREATED).body(nuevoCliente);
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }
```

En el método `createCliente` usamos `@RequestBody Cliente cliente` lo cual indica que el cuerpo de la solicitud HTTP (que se espera que sea un JSON) debe ser deserializado en un objeto Cliente.

En los métodos siguiente vemos diferentes combinaciones de `@PathVariable` y `@RequestBody`.

```
    @PutMapping("/{id}")
    public ResponseEntity<?> updateCliente(@PathVariable Long id, @RequestBody Cliente clienteDetails) {
        try {
            Cliente clienteActualizado = clienteService.updateCliente(id, clienteDetails);
            return ResponseEntity.ok(clienteActualizado);
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> deleteCliente(@PathVariable Long id) {
        try {
            boolean eliminado = clienteService.deleteCliente(id);
            if (eliminado) {
                return ResponseEntity.ok().body("Cliente eliminado exitosamente");
            } else {
                return ResponseEntity.notFound().build();
            }
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }

    @GetMapping("/email/{email}")
    public ResponseEntity<Cliente> getClienteByEmail(@PathVariable String email) {
        Optional<Cliente> cliente = clienteService.getClienteByEmail(email);
        if (cliente.isPresent()) {
            return ResponseEntity.ok(cliente.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @GetMapping("/nombre/{nombre}")
    public ResponseEntity<List<Cliente>> getClientesByNombre(@PathVariable String nombre) {
        List<Cliente> clientes = clienteService.getClientesByNombre(nombre);
        return ResponseEntity.ok(clientes);
    }

    @GetMapping("/telefono/{telefono}")
    public ResponseEntity<Cliente> getClienteByTelefono(@PathVariable String telefono) {
        Optional<Cliente> cliente = clienteService.getClienteByTelefono(telefono);
        if (cliente.isPresent()) {
            return ResponseEntity.ok(cliente.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @GetMapping("/ordenados")
    public ResponseEntity<List<Cliente>> getAllClientesOrderByFechaRegistro() {
        List<Cliente> clientes = clienteService.getAllClientesOrderByFechaRegistro();
        return ResponseEntity.ok(clientes);
    }
}
```

Podemos usar diferentes métodos de `ResponseEntity` de acuerdo al status que deberá devolver nuestra API.

▶️ Ejecutando la aplicación

Para probar la API que hemos creado será necesario correr nuestra aplicación, lo cuál podemos hacer ejecutando la clase `RestApplication`

```
@SpringBootApplication
public class RestApplication {

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

}
```

Esta clase contiene `@SpringBootApplication` la cual una anotación de conveniencia que agrupa tres anotaciones muy importantes:

- `@Configuration`: Marca la clase como una fuente de definiciones de beans.
- `@EnableAutoConfiguration`: Le dice a Spring Boot que comience a agregar beans basados en la configuración de la ruta de clases, otros beans y varias configuraciones de propiedades. Por ejemplo, si detecta spring-webmvc en la ruta de clases, configurará automáticamente un DispatcherServlet. Si detecta una base de datos como H2, la configurará automáticamente.
- `@ComponentScan`: Le dice a Spring que busque otros componentes, configuraciones y servicios en el paquete com.academy.rest (y sus subpaquetes), permitiéndole encontrar y registrar los controladores, servicios, repositorios, etc.

Finalmente dentro del método main, la línea `SpringApplication.run(...)` arranca la aplicación, inicia el contenedor de Spring y el servidor de aplicaciones embebido (generalmente Tomcat).

🚀 Consumiendo la API

Esta API se puede consumir de distintas maneras, ya sea utilizando Postman para realizar las consultas de los diferentes endpoints y probar que todo funciona correctamente o bien conectarla a un front-end que muestra gráficamente el resultado de nuestras peticiones. Para probar con Postman el endpoint `GET` que devuelve todos los clientes los pasos son los siguientes:

👷 Obtener Todos los Clientes (GET)

Este endpoint corresponde al método `listarClientes()`.

1. Crear una nueva petición haciendo clic en el botón "+" para abrir una nueva pestaña de petición.
2. Seleccionar el método HTTP eligiendo `GET` en el menú desplegable.
3. Introducir la URL en el campo correspondiente, escribiendo la dirección completa del endpoint.
    - http://localhost:8080/api/clientes
4. Enviar la petición haciendo clic en el botón azul **Send**.

Resultado esperado: En la parte inferior, en la pestaña Body, veremos una lista de objetos JSON, cada uno representando un cliente. Si la base de datos está vacía, se mostrará un array JSON vacío `[]`.

👤 Actualizar un Cliente (PUT)

Este endpoint corresponde al método actualizarCliente().

1. Crear una nueva petición y seleccionar PUT.
2. Introducir la URL del cliente que se quiere actualizar.
    - http://localhost:8080/api/clientes/1
3. Configurar el Cuerpo yendo a Body -> raw -> JSON e ingresamos los datos actualizados. Se deben incluir todos los campos.

```
{
    "id": 1,
    "nombre": "Ana Sofía",
    "apellido": "García López",
    "email": "anasofia.garcia@example.com"
}
```

4. Enviar la petición haciendo clic en Send.

Resultado esperado:

- Status Code: 200 OK.
- Body: Se mostrará el objeto JSON del cliente con los datos ya actualizados.

Y con esto hemos concluido satisfactoriamente nuestra API usando Spring y MySQL.