1. Preparativos

Descripción:

Este taller está diseñado para que los equipos trabajen en la creación de una API REST utilizando Spring Boot y MySQL. Los equipos ya conformados para el proyecto final colaborarán en cada etapa, desde la configuración inicial hasta la validación de funcionalidades utilizando Postman y la verificación de datos en MySQL Workbench.

Antes de comenzar, asegúrate de tener lo siguiente instalado:

- **Java JDK 17 o superior**.
- **IntelliJ IDEA** o cualquier otro IDE.
- **MySQL** y **MySQL Workbench**.
- **Postman** (para pruebas API).

2. Crear el Proyecto en Spring Boot

1. Ve a Spring Initializr.
2. Configura el proyecto:
    - **Project**: Maven Project
    - **Language**: Java
    - **Spring Boot Version**: 3.1.x
    - **Group**: com.tuempresa
    - **Artifact**: proyecto-demo
    - **Name**: proyecto-demo
    - **Packaging**: Jar
    - **Java version**: 17
3. Selecciona las dependencias:
    - **Spring Web**: Para crear APIs REST.
    - **Spring Data JPA**: Para manejar la base de datos con JPA.
    - **MySQL Driver**: Conector para MySQL.
    - **Spring Boot DevTools** (opcional).
4. Descarga y abre el proyecto en IntelliJ IDEA.

3. Configurar MySQL

Crea una base de datos en MySQL para que la aplicación pueda interactuar con ella:

1. Abre **MySQL Workbench** y ejecuta la siguiente consulta para crear la base de datos:

```
Unset
CREATE DATABASE proyectodemodb;
```

Luego, crea un usuario con permisos para esa base de datos:

```
Unset
CREATE USER 'usuario'@'localhost' IDENTIFIED BY 'contraseña'; GRANT ALL
PRIVILEGES ON proyectodemodb.* TO 'usuario'@'localhost'; FLUSH PRIVILEGES;
```

Ahora, la base de datos está lista para conectarse desde Spring Boot.

4. Configurar la conexión en Spring Boot

En el proyecto, abre el archivo `src/main/resources/application.properties` y agrega las propiedades de conexión a la base de datos MySQL:

```
Unset
# Configuración de la conexión a MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/proyectodemodb
spring.datasource.username=usuario
spring.datasource.password=contraseña
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Configuración de JPA e Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Explicación:

- **spring.datasource.url**: URL de conexión a la base de datos, donde `localhost` es el servidor MySQL local, `3306` es el puerto por defecto y `proyectodemodb` es la base de datos que acabamos de crear.

- **spring.datasource.username y password**: Usuario y contraseña para conectarse a MySQL.

- **spring.jpa.hibernate.ddl-auto=update**: Esto permite que Hibernate cree o actualice las tablas automáticamente.

- **spring.jpa.show-sql=true:** Muestra las consultas SQL que se ejecutan en la consola.

- **hibernate.dialect**: Indica el dialecto de SQL que Hibernate usará para MySQL.

5. Crear el Modelo

Crea una clase `Producto` en el paquete `model` que represente la entidad en la base de datos:

Java

```java
package com.tuempresa.proyectodemo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Autogenerar el ID
    private Long id;
    private String nombre;
    private Double precio;

    // Getters y Setters
    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 Double getPrecio() {
        return precio;
    }

    public void setPrecio(Double precio) {
        this.precio = precio;
    }
}
```

Explicación:

- **@Entity**: Indica que esta clase se va a mapear como una tabla en la base de datos.

- **@Id**: Define la columna `id` como la clave primaria.
- **@GeneratedValue**: Genera automáticamente un valor para la clave primaria. `GenerationType.IDENTITY` permite que la base de datos genere el valor del ID.

Cuando ejecutes la aplicación, Spring Boot creará automáticamente la tabla `Producto` en MySQL, basada en esta clase.

6. Crear el Repositorio

Crea una interfaz `ProductoRepository` en el paquete `repository`:

El archivo para la interfaz `ProductoRepository` debe ir en la siguiente ruta:

- **Ruta:** `src/main/java/com/tuempresa/proyectodemo/repository/ProductoRepository.java`

```java
package com.tuempresa.proyectodemo.repository;

import com.tuempresa.proyectodemo.model.Producto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {
}
```

Explicación:

- **JpaRepository:** Proporciona métodos CRUD (crear, leer, actualizar y eliminar) para la entidad **Producto** sin que tengas que implementarlos manualmente.

- **Long:** El tipo de dato del campo **id** de la entidad **Producto**.

7. Crear el Servicio

Ruta:
src/main/java/com/tuempresa/proyectodemo/service/ProductoService.java

Crea una clase **ProductoService** en el paquete **service**:

```java
package com.tuempresa.proyectodemo.service;

import com.tuempresa.proyectodemo.model.Producto;
import com.tuempresa.proyectodemo.repository.ProductoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class ProductoService {
```

```
    @Autowired
    private ProductoRepository productoRepository;

    public List<Producto> listarProductos() {
        return productoRepository.findAll();
    }

    public Producto agregarProducto(Producto producto) {
        return productoRepository.save(producto);
    }

    public Optional<Producto> obtenerProductoPorId(Long id) {
        return productoRepository.findById(id);
    }

    public void eliminarProducto(Long id) {
        productoRepository.deleteById(id);
    }
}
```

Explicación:

- **@Service:** Indica que esta clase es un servicio que contiene la lógica de negocio.
- **@Autowired:** Spring inyecta automáticamente una instancia de `ProductoRepository` para usar sus métodos.
- Métodos como `listarProductos()`, `agregarProducto()` y `eliminarProducto()` interactúan con el repositorio para realizar operaciones en la base de datos.

8. Crear el Controlador

Crea una clase `ProductoController` en el paquete `controller` para exponer los endpoints REST:

Java

```
package com.tuempresa.proyectodemo.controller;
```

```java
import com.tuempresa.proyectodemo.model.Producto;
import com.tuempresa.proyectodemo.service.ProductoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/productos")
public class ProductoController {

    @Autowired
    private ProductoService productoService;

    @GetMapping
    public List<Producto> listarProductos() {
        return productoService.listarProductos();
    }

    @PostMapping
    public Producto agregarProducto(@RequestBody Producto producto) {
        return productoService.agregarProducto(producto);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Producto> obtenerProducto(@PathVariable Long id) {
        Optional<Producto> producto =
productoService.obtenerProductoPorId(id);
        if (producto.isPresent()) {
            return ResponseEntity.ok(producto.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> eliminarProducto(@PathVariable Long id) {
        productoService.eliminarProducto(id);
        return ResponseEntity.noContent().build();
    }
}
```

Explicación:

- **@RestController**: Indica que esta clase define endpoints REST.
- **@RequestMapping("/api/productos")**: Define la ruta base de los endpoints de productos.
- **@GetMapping**: Para obtener todos los productos.
- **@PostMapping**: Para agregar un nuevo producto.
- **@GetMapping("/{id}")**: Para obtener un producto por su `id`.
- **@DeleteMapping("/{id}")**: Para eliminar un producto.

9. Ejecutar y Probar la Aplicación

1. Ejecuta el proyecto en IntelliJ (`ProyectoDemoApplication.java`).
2. Abre **Postman** y prueba los siguientes endpoints:
    - **GET**: `http://localhost:8080/api/productos` - Para listar productos.
    - **POST**: `http://localhost:8080/api/productos` - Para crear un producto (envía un JSON como este en el cuerpo):

```
Unset
{
  "nombre": "Producto 1",
  "precio": 100.0
}
```

10. Ver los Resultados en MySQL Workbench

1. Abre **MySQL Workbench** y conéctate a tu base de datos `proyectodemodb`.
2. Ejecuta una consulta para ver la tabla y los productos insertados:

```
Unset
SELECT * FROM producto;
```