CRUD SpringBoot MySQL JPA Lombok

# TUTORIAL JAVA - Crear CRUD con Spring Boot - Parte1

<https://youtu.be/Blov4JXG8nA?t=8>

<https://youtu.be/bHL6Zyek41s>

Muy lindo tutorial, claro, rápido. La pronunciación del inglés de este muchacho es mala. Por favor, aprendan de Java, pero no de inglés.

# Conceptos usados

* Base de datos MySQL.
* Spring JPA.
* API RESTFul.
* CRUD.

Es todo backend. No hay frontend. Las operaciones se hacen por medio de PostMan.

# PostMan

Para consumir los servicios REST vamos a usar una herramienta que se llama PostMan. Hay otras opciones. Vamos a la página:

<https://www.postman.com/downloads/>

y descargamos la versión apropiada. Hasta donde yo sé, no se instala para todo el sistema, sino para el usuario. Esto significa que estará en un lugar como:

C:\Users\*USUARIO*\AppData\Local\Postman\Postman.exe

# IDE

Usa Spring Tool Suite. Yo uso VS Code.

# Crear nuevo proyecto de Java

Java: Create Java Project…

Spring Boot

Maven Project

2.6.6

Java

tup

crud

Jar

11

# Dependencias

MySQL Driver

Spring Data JPA

Spring Boot DevTools

Lombok Developer Tools

Spring Web

# Base de datos y usuario

Usa una base de datos de animes. Ver anime.sql en este mismo directorio. Le cambié el set de caracteres, el ordenamiento, arreglé algunas desprolijidades, agregué la creación del usuario con password, para no dejar el root.

# application.properties

<https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/howto-data-access.html>

En este archivo configuramos la conexión a la base de datos.

spring.application.name=crud

server.port=8080

spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://${MYSQL\_HOST:localhost}:3306/anime

spring.datasource.username=cruduser

spring.datasource.password=crudPassword

# https://stackoverflow.com/questions/50322550/hibernate-dialect-for-mysql-8

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

# pom.xml

El nodo de dependencias del archivo pom.xml, en mi pc quedó de la siguiente manera:

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

            <scope>runtime</scope>

        </dependency>

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-collections4 -->

        <dependency>

            <groupId>org.apache.commons</groupId>

            <artifactId>commons-collections4</artifactId>

            <version>4.4</version>

        </dependency>

# Paquetes y clases

Cambié los nombres, para hacerlos más consistentes con lo que venimos haciendo. Esto no es una regla, sino una costumbre o convención.

Escribo explícitamente la palabra clave public antes de los métodos, porque a partir de Java 9 las interfaces pueden tener métodos privados. No cuesta nada, y elimina la posibilidad de que se entienda mal.

# src\main\java\tup\crud\CrudApplication.java

package tup.crud;

/\*

Esta es la clase principal de la aplicación.

Vemos que solo tiene las anotaciones de SpringBoot.

\*/

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class CrudApplication {

    public static void main(String[] args) {

        SpringApplication.run(CrudApplication.class, args);

    }

}

# src\main\java\tup\crud\controllers\AnimeController.java

package tup.crud.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import tup.crud.models.Anime;

import tup.crud.services.AnimeServiceImpl;

@RestController

// La URL que vaya en la anotación habrá que agregarla detrás del puerto :8080 en todas las llamadas

// a esta aplicación. Por ejemplo @RequestMapping("/anime") resultaría en lo siguiente:

// localhost:8080/anime....   y detrás de esto habría que agregar el resto de la URL para ser RESTful.

// En este caso, no necesitamos nada, y queda simplemente localhost:8080

@RequestMapping("")

public class AnimeController {

    @Autowired

    private AnimeServiceImpl animeService;

    @GetMapping("/all")

    public ResponseEntity<List<Anime>> getAll() {

        return new ResponseEntity<List<Anime>>(this.animeService.getAll(), HttpStatus.OK);

    }

    @PostMapping("/save")

    public ResponseEntity<Anime> save(@RequestBody Anime anime) {

        return ResponseEntity.status(HttpStatus.CREATED).body(this.animeService.save(anime));

    }

    @PostMapping("/delete/{id}")

    public ResponseEntity<Anime> delete(@PathVariable Long id) {

        this.animeService.deleteById(id);

        return ResponseEntity.ok().build();

    }

}

# src\main\java\tup\crud\models\Anime.java

package tup.crud.models;

import java.time.LocalDateTime;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

import lombok.Data;

// https://www.educative.io/edpresso/what-is-the-data-annotation-in-lombok

// The @Data annotation does the following work:

//   It generates the getter methods for all the fields.

//   It generates the setter methods for all the non-final fields.

//   It generates the toString() method implementation.

//   It generates appropriate equals() and hashCode() implementations, involving the fields of class.

//   It generates a constructor that initializes all the final fields, as well as all the

//      non-final fields with no initializer that have been marked with @NonNull, in order

//      to ensure that the field is never null.

@Data

// La anotación @Entity de JPA indica a la plataforma que inyecte a esta clase

// las dependencias necesarias para ser una entidad persistente. Para eso, la

// clase necesita por ejemplo un constructor,

// pero de eso ya se ocupó la anotación @Data de Lombok.

@Entity

// Permite especificar el nombre de la tabla en la base de datos que JPA usará

// en las operaciones de persistencia de esta entidad.

// También podemos dar el nombre de la base de datos.

// https://www.baeldung.com/jpa-entities#table

@Table(name = "anime")

public class Anime {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    @Column(name = "id")

    // El tipo Long del campo id viene forzado por la interfaz genérica

    // CrudRepository.

    private Long id;

    @Column(name = "nombre")

    private String nombre;

    @Column(name = "anio")

    private String anio;

    @Column(name = "fecha\_creacion")

    private LocalDateTime fecha\_creacion;

}

Vemos que hay varias anotaciones. Este artículo de Baeldung cubre todas las anotaciones:

<https://www.baeldung.com/jpa-entities>

Este otro es más amigable, y cubre todo lo que necesitamos por ahora:

<https://www.paradigmadigital.com/dev/proyecto-lombok-facilitame-la-vida/>

## Anotación @Data

@Data es la anotación de acceso directo que agrupa las características de @ToString, @EqualsAndHashCode, @Getter/@Setter y @RequiredArgsConstructor juntas: en otras palabras, @Data genera todos los estándares que normalmente se asocian con POJOS y beans:

* Getters para todos los campos.
* Setters para todos los campos no finales y las implementaciones toString.
* Equals y hashCode que involucran los campos de la clase.
* Un constructor que inicializa todos los campos finales.
* Así como todos los campos no finales sin inicializador que haya sido marcado con @NonNull, para garantizar que el campo nunca sea nulo.

## Anotación @Entity

<https://stackoverflow.com/a/29333628>

La anotación @Entity define que una clase se puede asignar a una tabla. Y eso es todo, es solo un marcador, como por ejemplo la interfaz Serializable. ¿Y por qué la anotación @Entity es obligatoria? ... bueno, es la forma en que se diseña JPA.

Cuando crea una nueva entidad, debe hacer al menos dos cosas:

* anote la clase con @Entity,
* cree un campo de identificación y anótelo con @Id.

Cualquier otra cosa es opcional, por ejemplo, el nombre de la tabla se deriva del nombre de la clase de entidad (y, por lo tanto, la anotación @Table puede ser opcional), las columnas de la tabla se derivan de variables de entidades (y, por lo tanto, la anotación @Column puede ser opcional), y así sucesivamente... JPA está tratando de proporcionar un comienzo rápido y fácil para los desarrolladores que desean aprender

## Anotación @Table

<https://stackoverflow.com/a/37733064>

La anotación @Table le permite especificar los detalles de la tabla que se utilizará para conservar la entidad en la base de datos. La anotación @Table proporciona cuatro atributos, lo que le permite cambiar el nombre de la tabla, su catálogo y su esquema, y aplicar restricciones únicas en las columnas de la tabla.

## Anotaciones @Id y @GeneratedValue

<https://stackoverflow.com/questions/20603638/what-is-the-use-of-annotations-id-and-generatedvaluestrategy-generationtype>

La anotación @Id se hereda de javax.persistence.Id. Indica que el campo de miembro a continuación es la clave principal de la entidad actual. Por lo tanto, Hibernate y Spring pueden hacer algunos trabajos de reflexión basados en esta anotación.

La anotación @GeneratedValue es para configurar la forma de incremento de la columna (campo) especificada. Por ejemplo, al usar Mysql, puede especificar auto\_increment en la definición de la tabla para que sea autoincremental y luego usar

@GeneratedValue(strategy = GenerationType.IDENTITY)

en el código Java para indicar que también quiere usar esta estrategia del lado del servidor de la base de datos. Además, puede cambiar el valor de esta anotación para adaptarse a diferentes requisitos.

# src\main\java\tup\crud\repositories\AnimeRepository.java

package tup.crud.repositories;

import org.springframework.data.repository.CrudRepository;

import tup.crud.models.Anime;

// CrudRepository

// https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

// Interface for generic CRUD operations on a repository for a specific type.

// Publica los métodos count(), delete(), deleteAll(), deleteAllBiId(), deleteById(), existsById(),

// findAll(), findAllById(), findById(), save() y saveAll().

// Vemos que AnimeRepository es una interfaz. Como tal, no implementa ningún método.

// No publica ni métodos ni propiedades. Se limita

// a extender CrudRepository, que es también una interfaz: publica pero no implementa.

// En otras palabras, no hemos implementado ni métodos ni propiedades.

// La magia ocurre en la clase AnimeServiceImpl, que tiene un campo de tipo

// AnimeRepository, y ese campo tiene la anotación @Autowired. Esa anotación es la

// que se ocupa de la inyección de dependencias.

// La interfaz CrudRepository es genérica en el tipo Anime. La clase Anime está en el

// paquete models.

public interface AnimeRepository extends CrudRepository<Anime, Long> {

}

# src\main\java\tup\crud\services\AnimeService.java

package tup.crud.services;

/\* AnimeService no es una clase, sino una interfaz. Publica métodos, pero

 \* no implementa nada. Esto se llama interfaz contractual. Si vamos a

 \* organizar nuestra aplicación publicando en una o más APIs los

 \* servicios o microservicios que queremos que nuestros clientes consuman,

 \* esta interfaz es lo que vamos a dar a conocer a nuestros clientes.

 \* Todas las implementaciones están en la clase AnimeServiceImpl.

 \* Están encapsuladas, ocultas a los clientes. En realidad, a los

 \* clientes no deberían interesarles los detalles de la implementación,

 \* en tanto en cuanto cumplamos a rajatabla con lo que la interfaz contractual

 \* promete.

 \*/

import java.util.List;

import tup.crud.models.Anime;

public interface AnimeService {

    public List<Anime> getAll();

    public Anime save(Anime anime);

    public void deleteById(Long id);

}

# src\main\java\tup\crud\services\AnimeServiceImpl.java

package tup.crud.services;

/\* La clase AnimeServiceImpl encapsula la implementación de todo lo que

 \* promete la interfaz contractual AnimeService.

 \* Esta clase no es para ser mostrada a los consumidores de los servicios.

 \* Podemos modificar a gusto esta clase, y siempre que nuestra implementación

 \* cumpla lo que promete la interfaz contractual, nadie puede objetar, porque no

 \* vamos a incurrir en ninguna incompatibilidad.

 \*/

import java.time.LocalDateTime;

import java.util.Iterator;

import java.util.List;

// public class IteratorUtils extends Object

// Provides static utility methods and decorators for Iterator instances. The implementations

// are provided in the iterators subpackage.

// https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/IteratorUtils.html

import org.apache.commons.collections4.IteratorUtils;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import tup.crud.models.Anime;

import tup.crud.repositories.AnimeRepository;

@Service

public class AnimeServiceImpl implements AnimeService {

    // https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html

    @Autowired

    private AnimeRepository animeRepository;

    @Override

    public List<Anime> getAll() {

        // animeRepository es de tipo AnimeRepository, que es una interfaz.

        // AnimeRepository a su vez extiende la interfaz CrudRepository.

        // La interfaz CrudRepository es genérica. En nuestro caso, la declaración es

        // CrudRepository<Anime, Long>. Anime es el tipo sobre el que implementa la

        // programación genérica, y Long es el tipo que se requiere para el campo Id.

        // Esto significa que el repositorio almacena objetos (instancias) del tipo

        // Anime.

        // La anotación @Autowired está justo antes de la declaración de

        // animeRepository,

        // de modo que afecta solo a este campo. Esta anotación elimina la necesidad de

        // implementar constructores, getters y setters. No necesitamos hacerlo

        // nosotros, sino que la plataforma lo hace automáticamente.

        // El método findAll() retorna todas las instancias de la clase Anime

        // contenidas en el repositorio. O sea, trae todos los registros de la

        // tabla asociada con el tipo Anime. Pero los retorna de un modo preciso,

        // como un objeto de tipo Iterable<Anime>.

        // https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html?is-external=true

        // La interfaz Iterable hace lo que su nombre indica. En particular, el método

        // iterator() retorna un objeto de tipo Iterator<Anime>.

        // La interfaz Iterator permite iterar una colección.

        // https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

        // Y así llegamos a la declaración Iterator<Anime> animeIterator.

        // Este tipo tiene solo 4 métodos, y ninguno de ellos nos sirve.

        //

        Iterator<Anime> animeIterator = this.animeRepository.findAll().iterator();

        List<Anime> animeList = IteratorUtils.toList(animeIterator);

        return animeList;

    }

    @Override

    public Anime save(Anime anime) {

        anime.setFecha\_creacion(LocalDateTime.now());

        return this.animeRepository.save(anime);

    }

    @Override

    public void deleteById(Long id) {

        this.animeRepository.deleteById(id);

    }

}

Esta clase usa el método IteratorUtils.toList(), que está en una biblioteca de Apache. La dependencia correspondiente está en pom.xml. El autor propone otra biblioteca, pero a mí me gusta más esta variante.

# Pruebas

## Listar todos

Esto es un request de tipo GET. Lo podemos hacer desde el browser, y también desde PostMan.

Una vez arrancada la aplicación, en el browser poner:

<http://localhost:8080/anime/all>

Si este mismo request lo enviamos por PostMan, la respuesta aparece formatteada mucho más linda.

## Grabar nuevo registro

Es de tipo POST. No lo podemos hacer con el browser, al menos no fácilmente.

En PostMan crear un request de tipo POST, URL: http://localhost:8080/anime/save

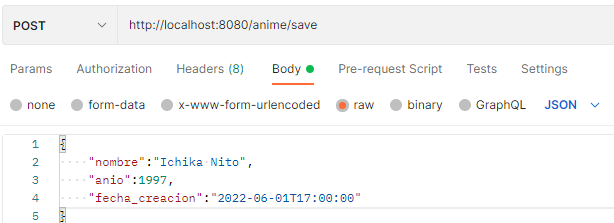
En el Body del request seleccionar raw, tipo JSON, y poner:

{

    "nombre":"Ichika Nito",

    "anio":1997,

    "fecha\_creacion":"2022-06-01T17:00:00"

}

PostMan nos devuelve Status 201 Created. Si repetimos la consulta anterior, podemos comprobar que se agregó a la base de datos.

## Borrar registro por id

En este ejemplo, 15 es el id del registro creado justo en el paso anterior. PostMan nos retorna Status 200 Ok.

Si consultamos la base de datos, veremos que fue borrado.