CRUD Spring Boot, Angular, MySql, Docker

* Funciona todo.
* He seguido los tutoriales como la base, pero mi proyecto está muy cambiado respecto del que propone el autor.
* Agregué una cantidad de comentarios.
* Cambié a una estructura de paquetes más estándar.
* La implementación que hace el autor del método subscribe() está deprecada. La cambié por la implementación que ahora se recomienda.
* JPA puede crear la base de datos y las tablas, y cargar datos. Por ahora nos quedamos con el concepto, pero no lo vamos a ver en detalle.

**CORS with Spring**

<https://www.baeldung.com/spring-cors>

**Spring Boot y la anotación @CrossOrigin**

<https://ricardogeek.com/spring-boot-y-la-anotacion-crossorigin/>

**#1 CRUD BASICO Angular, Spring boot y MySql: creación de proyecto y modelo en backend**

<https://youtu.be/4VWQwrM1voU>

**#2 CRUD BASICO Angular, Spring boot y MySql: service y controller**

<https://youtu.be/icqYtxxSpEY>

**#3 CRUD BASICO Angular, Spring boot y MySql: Angular Creacion de proyecto y service**

<https://youtu.be/3divgzHPX5g>

**#4 CRUD BASICO Angular, Spring boot y MySql: diseño de formularios con bootstrap y rutas**

<https://youtu.be/Z6zcyenQSwY>

**Como Dockerizar una aplicación en Angular**

<https://youtu.be/CIKj9ftIMY8>

CRUD básico (create-read-update-delete) full-stack con Spring Boot como backend, Angular 12 como frontend y MySQL como base de datos.

El proyecto es un administrador de tareas, o tasks.

Arranco WampServer, para la base de datos.

Base de datos: tareas

El archivo Database tareas.sql tiene el script para crear la base de datos. Notar que no es igual al que usa el autor. Le hice unos cambios.

En MySQL no existe un tipo de dato bool o boolean. Los acepta como provenientes de otros DBMS y los representa como tiny int(1).

<https://stackoverflow.com/a/42147995/2740402>

For the record, the spring.jpa.hibernate.ddl-auto property is Spring Data JPA specific and is their way to specify a value that will eventually be passed to Hibernate under the property it knows, hibernate.hbm2ddl.auto.

The values create, create-drop, validate, and update basically influence how the schema tool management will manipulate the database schema at startup.

For example, the update operation will query the JDBC driver's API to get the database metadata and then Hibernate compares the object model it creates based on reading your annotated classes or HBM XML mappings and will attempt to adjust the schema on-the-fly.

The update operation for example will attempt to add new columns, constraints, etc but will never remove a column or constraint that may have existed previously but no longer does as part of the object model from a prior run.

Typically in test case scenarios, you'll likely use create-drop so that you create your schema, your test case adds some mock data, you run your tests, and then during the test case cleanup, the schema objects are dropped, leaving an empty database.

In development, it's often common to see developers use update to automatically modify the schema to add new additions upon restart. But again understand, this does not remove a column or constraint that may exist from previous executions that is no longer necessary.

In production, it's often highly recommended you use none or simply don't specify this property. That is because it's common practice for DBAs to review migration scripts for database changes, particularly if your database is shared across multiple services and applications.

# Aplicación Spring Boot

La aplicación está lista para usar. Abrimos con VS Code la carpeta 30-crud-spring-boot-angular-mysql-docker\backend.

# src\main\resources\application.properties

Editar application.properties de modo que quede:

# Debe ser igual al artifactId del pom

spring.application.name=backend

server.port=8080

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

spring.datasource.url=jdbc:mysql://mysqldb:3306/tareas

spring.datasource.username=tareasuser

spring.datasource.password=tareaspassword

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

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

# https://stackoverflow.com/a/42147995/2740402

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

Notar que el host de la base de datos es el servicio de MySQL.

# sql\Database tareas.sql

Este archivo tiene el script para crear la base de datos y el usuario.

DROP DATABASE IF EXISTS tareas;

CREATE DATABASE IF NOT EXISTS tareas DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;

DROP USER IF EXISTS `tareasuser`;

CREATE USER IF NOT EXISTS `tareasuser` @`%` IDENTIFIED BY 'tareaspassword';

GRANT USAGE ON \*.\* TO `tareasuser` @`%`;

GRANT ALL PRIVILEGES ON `tareas`.\* TO 'tareasuser ' @' % ';

USE tareas;

DROP TABLE IF EXISTS tarea;

CREATE TABLE IF NOT EXISTS tarea (

  id int UNSIGNED NOT NULL AUTO\_INCREMENT,

  tarea varchar(200) NOT NULL,

  finalizado tinyint(1) NOT NULL,

  PRIMARY KEY (id)

) ENGINE = MyISAM AUTO\_INCREMENT = 3 DEFAULT CHARSET = utf8mb4 COLLATE = utf8mb4\_0900\_ai\_ci;

INSERT INTO tarea (id, tarea, finalizado)

VALUES (1, 'Reunión diaria | 08:30', 0),

  (2, 'Coordinar reunión ActivaLogic', 0);

# src\main\java\gga\backend\BackendApplication.java

package gga.backend;

/\*

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 BackendApplication {

    public static void main(String[] args) {

        SpringApplication.run(BackendApplication.class, args);

    }

}

# src\main\java\gga\backend\controllers\TareaController.java

package gga.backend.controllers;

import java.util.List;

import java.util.Optional;

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

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

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

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.PutMapping;

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

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

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

import gga.backend.models.Tarea;

import gga.backend.services.TareaService;

// Annotation @Component

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html

// Indicates that an annotated class is a "component".

// Such classes are considered as candidates for

// auto-detection when using annotation-based

// configuration and classpath scanning.

// Factory pattern, o patrón factoría.

// https://en.wikipedia.org/wiki/Factory\_method\_pattern

// Spring usa el patrón factoría. Para marcar una interfaz o clase como

// factoría,

// se usa alguna de las anotaciones @Controller, @Service, @Repository o

// @RestController.

// Annotation @Controller

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html

// Indicates that an annotated class is a

// "Controller" (e.g. a web controller).

// This annotation serves as a specialization

// of @Component, allowing for implementation

// classes to be autodetected through classpath

// scanning. It is typically used in combination

// with annotated handler methods based on

// the @RequestMapping annotation.

// Annotation @Service

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html

// Indicates that an annotated class is a "Service":

// "an operation offered as an interface that stands

// alone in the model, with no encapsulated state."

// Originally defined by Domain-Driven Design (Evans, 2003).

// This annotation serves as a specialization of @Component,

// allowing for implementation classes to be autodetected

// through classpath scanning.

// Annotation @Repository

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html

// Indicates that an annotated class is a "Repository":

// "a mechanism for encapsulating storage, retrieval,

// and search behavior which emulates a collection of objects".

// Originally defined by Domain-Driven Design (Evans, 2003).

// Teams implementing traditional Java EE patterns such as

// "Data Access Object" may also apply this stereotype to

// DAO classes, though care should be taken to understand

// the distinction between Data Access Object and

// DDD-style repositories before doing so.

// A class thus annotated is eligible for Spring

// DataAccessException translation when used in conjunction

// with a PersistenceExceptionTranslationPostProcessor.

// The annotated class is also clarified as to its role

// in the overall application architecture for the purpose

// of tooling, aspects, etc.

// As of Spring 2.5, this annotation also serves as a

// specialization of @Component, allowing for implementation

// classes to be autodetected through classpath scanning.

// Annotation @RestController

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html

// A convenience annotation that is itself annotated

// with @Controller and @ResponseBody.

// Types that carry this annotation are treated as

// controllers where @RequestMapping methods

// assume @ResponseBody semantics by default.

// Annotation @ResponseBody

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html

// Annotation that indicates a method return value

// should be bound to the web response body.

// Supported for annotated handler methods.

// As of version 4.0 this annotation can also be

// added on the type level in which case it is

// inherited and does not need to be added

// on the method level.

@RestController

// Vamos a tener un frontend en Angular. Los requests de

// ese frontend vendrán desde el puerto :4200.

// Al llegar al puerto :8080 serán vistos como provenientes

// de otro dominio. Y vamos a tener problemas con CORS.

// Para prevenir esto, tenemos que indicar los orígenes

// de los cuales queremos recibir requests.

// https://www.baeldung.com/spring-cors

// @CrossOrigin(origins = "http://localhost:4200")

@CrossOrigin(origins = "\*")

// Annotation @RequestMapping

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html

// Annotation for mapping web requests onto methods

// in request-handling classes with flexible

// method signatures.

// Both Spring MVC and Spring WebFlux support

// this annotation through a RequestMappingHandlerMapping

// and RequestMappingHandlerAdapter in their

// respective modules and package structure. For the

// exact list of supported handler method arguments

// and return types in each, please use the reference

// documentation links below:

// Spring MVC Method Arguments and Return Values

// Spring WebFlux Method Arguments and Return Values

// Note: This annotation can be used both at the class

// and at the method level. In most cases, at the method

// level applications will prefer to use one of the HTTP

// method specific variants @GetMapping, @PostMapping,

// @PutMapping, @DeleteMapping, or @PatchMapping.

// La URL que vaya en la anotación habrá que agregarla detrás

// del puerto :8080 en todas las llamadas a esta aplicación.

// Por ejemplo @RequestMapping("/asdf")

// resultaría en lo siguiente:

// localhost:8080/asdf.... y detrás de esto habría que

// agregar el resto de la URL para ser RESTful.

// En este caso, queda como localhost:8080/api

@RequestMapping("/api")

public class TareaController {

    // Annotation @Autowired

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

    // Marks a constructor, field, setter method, or config

    // method as to be autowired by Spring's dependency

    // injection facilities.

    // Una vez que los objetos estan creados, la anotación

    // Spring @Autowired se encarga de construir las

    // ligazones entre los distintos elementos.

    @Autowired

    private TareaService tareaService;

    @GetMapping("/tareas")

    public List<Tarea> listar() {

        return tareaService.findAll();

    }

    @PostMapping("/tareas")

    public Tarea guardar(@RequestBody Tarea tarea) {

        return tareaService.save(tarea);

    }

    @GetMapping("/tareas/{id}")

    public Optional<Tarea> findByID(@PathVariable Long id) {

        return tareaService.findById(id);

    }

    @PutMapping("/tareas/{id}")

    public Tarea modificar(@RequestBody Tarea tarea, @PathVariable Long id) {

        Tarea tareaActual = tareaService.findById(id).get();

        tareaActual.setTarea(tarea.getTarea());

        tareaActual.setFinalizado(tarea.isFinalizado());

        return tareaService.save(tareaActual);

    }

    @DeleteMapping("/tareas/{id}")

    public void eliminar(@PathVariable Long id) {

        tareaService.deleteById(id);

    }

}

# src\main\java\gga\backend\models\Tarea.java

package gga.backend.models;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

// @Entity

// https://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html

// Identifies a class as an entity class. Specifies that the class is an entity.

// This annotation is applied to the entity class.

// https://thorben-janssen.com/key-jpa-hibernate-annotations/

// Define an Entity Class

// JPA entities don’t need to implement any interface or extend a superclass.

// They are simple POJOs (Plain Old Java Object). But you still need to

// identify a class as an entity class, and you might want to adapt

// the default table mapping.

@Entity

// @Table

// https://docs.oracle.com/javaee/7/api/javax/persistence/Table.html

// Specifies the primary table for the annotated entity.

// Additional tables may be specified using SecondaryTable

// or SecondaryTables annotation.

// If no Table annotation is specified for an entity

// class, the default values apply.

// By default, each entity class maps a database table

// with the same name in the default schema of your database.

// You can customize this mapping using the name, schema,

// and catalog attributes of the @Table annotation.

// The name attribute enables you to change the name of the

// database table which your entity maps.

// The schema attribute specifies the name of the database

// schema in which the table is located.

// And the catalog attribute describes the name of the

// database catalog that stores the metadata information

// of the table.

// The @Table annotation also defines 2 attributes that

// enable you to influence the generation of the database

// table. These are called indexes and uniqueConstraints.

// I don’t recommend to use them. External script and tools

// like Liquibase or Flyway are a much better option to

// create and update your database.

@Table(name = "tarea")

public class Tarea {

    // @Id

    // https://docs.oracle.com/javaee/7/api/javax/persistence/Id.html

    // Specifies the primary key of an entity. The field or property to

    // which the Id annotation is applied should be one of the following

    // types: any Java primitive type; any primitive wrapper type;

    // String; java.util.Date; java.sql.Date; java.math.BigDecimal;

    // java.math.BigInteger.

    // The mapped column for the primary key of the entity is assumed

    // to be the primary key of the primary table.

    @Id

    // @GeneratedValue

    // https://docs.oracle.com/javaee/7/api/javax/persistence/GeneratedValue.html

    // Provides for the specification of generation strategies for the values

    // of primary keys.

    // The GeneratedValue annotation may be applied to a primary key

    // property or field of an entity or mapped superclass in conjunction

    // with the Id annotation. The use of the GeneratedValue annotation

    // is only required to be supported for simple primary keys. Use of

    // the GeneratedValue annotation is not supported for derived primary keys.

    // When we’re talking about primary keys, we also need to talk about sequences

    // and auto-incremented database columns.

    // These are the 2 most common database features to generate unique primary key

    // values.

    // If you annotate your primary key attribute with the @GeneratedValue

    // annotation, you can use a database sequence by setting the strategy

    // attribute to GenerationType.SEQUENCE. Or, if you want to use an

    // auto-incremented

    // database column to generate your primary key values, you need to set

    // the strategy to GenerationType.IDENTITY.

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    // If no Column annotation is specified, the primary key column

    // name is assumed to be the name of the primary key property or field.

    // JPA and Hibernate require you to specify at least one primary key

    // attribute for each entity.

    @Column(name = "id")

    private Long id;

    // @Column

    // https://docs.oracle.com/javaee/7/api/javax/persistence/Column.html

    // It is an optional annotation.

    // Specifies the mapped column for a persistent property or field.

    // If no Column annotation is specified, the default values apply.

    // You can use the name attribute to specify the name of the

    // database column which the entity attribute map.

    // The attributes updatable and insertable enable you to exclude

    // the attribute from insert or update statements.

    @Column(name = "tarea")

    private String tarea;

    @Column(name = "finalizado")

    private boolean finalizado;

    /\*\*

     \* JPA exige este constructor default, sin argumentos.

     \* Si no está, se produce una excepción.

     \*/

    public Tarea() {

    }

    public Tarea(String tarea, boolean finalizado) {

        this.tarea = tarea;

        this.finalizado = finalizado;

    }

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getTarea() {

        return tarea;

    }

    public void setTarea(String tarea) {

        this.tarea = tarea;

    }

    public boolean isFinalizado() {

        return finalizado;

    }

    public void setFinalizado(boolean finalizado) {

        this.finalizado = finalizado;

    }

}

# src\main\java\gga\backend\repositories\TareaRepository.java

package gga.backend.repositories;

import org.springframework.data.repository.CrudRepository;

import gga.backend.models.Tarea;

// El paquete repositories está pensado para la persistencia.

// Se ocupa de la tabla que representa a esta entidad en la

// base de datos. No ofrece ningún servicio. Eso tiene que

// ver con el principio de separación de responsabilidades.

// 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 TareaRepository 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 TareaServiceImpl, que tiene un campo de tipo

// TareaRepository, 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 Tarea. La clase Tarea está en el

// paquete models.

// Esto es un mecanismo de inyección de dependencias. La interfaz TareaRepository

// tiene inyectada la dependencia de la clase Tarea, gracias a esta

// declaración en el extends.

public interface TareaRepository extends CrudRepository<Tarea, Long> {

}

# src\main\java\gga\backend\services\TareaService.java

package gga.backend.services;

// El paquete services expone los servicios que los clientes consumen.

// Solo se ocupa de eso. En particular, no se ocupa de la persistencia.

// TareaService 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 TareaServiceImpl.

// 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.

// Los nombres de los métodos que expone son iguales a los de la

// interfaz CrudRepository. Pero no la extiende. Si esta interfaz

// extendiera CrudRepository, terminaríamos inyectando una dependencia

// que no necesitamos en este paquete, porque está en el paquete

// repositories.

// El instinto me dice que es mejor usar los mismos nombres con las

// mismas signaturas. Creo que es mejor estilo. Pero no es obligatorio.

import java.util.List;

import java.util.Optional;

import gga.backend.models.Tarea;

public interface TareaService {

    public void deleteById(Long id);

    public List<Tarea> findAll();

    public Optional<Tarea> findById(Long id);

    public Tarea save(Tarea tarea);

}

# src\main\java\gga\backend\services\TareaServiceImpl.java

package gga.backend.services;

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

 \* promete la interfaz contractual TareaService.

 \* 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.util.List;

import java.util.Optional;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import gga.backend.models.Tarea;

import gga.backend.repositories.TareaRepository;

// @Service Annotation

// https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html

// Indicates that an annotated class is a "Service":

// "an operation offered as an interface that stands

// alone in the model, with no encapsulated state."

// Originally defined by Domain-Driven Design (Evans, 2003).

// This annotation serves as a specialization of @Component,

// allowing for implementation classes to be autodetected

// through classpath scanning.

// In an application, the business logic resides within the service

// layer so we use the @Service Annotation to indicate that a

// class belongs to that layer. It is also a specialization

// of @Component Annotation like the @Repository Annotation.

// One most important thing about the @Service Annotation

// is it can be applied only to classes. It is used to mark

// the class as a service provider. So overall @Service

// annotation is used with classes that provide some business

// functionalities. Spring context will autodetect these

// lasses when annotation-based configuration and classpath

// scanning is used.

// En resumen, esta clase está anotada con @Service porque

// esta clase, de hecho, expone los servicios, entendidos

// como una operación que un cliente consume, y además

// esta clase no tiene un estado encapsulado.

// Notar que la única propiedad que tiene es tareaRepository,

// que está anotada como @Autowired

@Service

public class TareaServiceImpl implements TareaService {

    // tareaRepository es de tipo TareaRepository, que es una interfaz.

    // TareaRepository a su vez extiende la interfaz CrudRepository.

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

    // CrudRepository<Tarea, Long>. Tarea 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 Tarea.

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

    // tareaRepository, de modo que afecta solo a este campo.

    // Esto es una inyección de dependencia.

    // Esta anotación elimina la necesidad de implementar

    // constructores, getters y setters. No necesitamos hacerlo

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

    // O sea que el campo tareaRepository tendrá automáticamente implementados

    // todos los métodos que necesite, que son los de CrudRepository.

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

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

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

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

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

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

    // La interfaz Iterable hace lo que su nombre indica.

    // List extiende Collection, y Collection extiende Iterable.

    // Por eso podemos castear Iterable<Tarea> en List<Tarea>

    //

    @Autowired

    private TareaRepository tareaRepository;

    /\*

     \* @Override La anotación @Override informa al compilador

     \* que el elemento está destinado a override un elemento

     \* declarado en una superclase.

     \* Si bien no es necesario usar esta anotación al

     \* override un método, ayuda a evitar errores.

     \* Si un método marcado con @Override no puede

     \* override correctamente un método en una de sus

     \* superclases, el compilador genera un error.

     \*/

    @Override

    /\*

     \* https://stackoverflow.com/a/54326467/2740402

     \* Generalmente, la anotación @Transactional

     \* se escribe en el nivel de servicio. Se utiliza

     \* para combinar más de una escritura en una base

     \* de datos como una sola operación atómica.

     \* Cuando alguien llama al método anotado con

     \*

     \* @Transactional, se ejecutan todas o ninguna

     \* de las escrituras en la base de datos. En el

     \* caso de operaciones de lectura no es útil y

     \* tampoco lo es en el caso de una sola escritura

     \* atómica.

     \*/

    @Transactional(readOnly = false)

    public void deleteById(Long id) {

        tareaRepository.deleteById(id);

    }

    @Override

    public List<Tarea> findAll() {

        return (List<Tarea>) tareaRepository.findAll();

    }

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

    // Notar el tipo de retorno: Optional<Tarea>

    // Viene directamente de la plataforma.

    @Override

    public Optional<Tarea> findById(Long id) {

        return tareaRepository.findById(id);

    }

    @Override

    @Transactional(readOnly = false)

    public Tarea save(Tarea tarea) {

        return tareaRepository.save(tarea);

    }

}

# backend\Dockerfile

Creamos un archivo Dockerfile en la carpeta 30-crud-spring-boot-angular-mysql-docker\backend y le pegamos el texto:

FROM eclipse-temurin

COPY target/\*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

# Frontend en Angular

Abrimos la carpeta 30-crud-spring-boot-angular-mysql-docker\frontend con VS Code.

Recordar que el frontend va a tratar de comunicarse con el backend, de modo que es necesario que el servicio de Spring Boot esté corriendo, y que WampServer esté también corriendo, porque es lo que estamos usando para habilitar el servicio de MySQL.

# src\index.html

<!doctype html>

<html lang="es">

<head>

  <meta charset="utf-8">

  <!-- Este título aparece en la pestaña del browser -->

  <title>Frontend</title>

  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">

  <link rel="icon" type="image/x-icon" href="assets/favicon.ico">

  <!-- CSS only -->

  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet"

    integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmVor" crossorigin="anonymous">

  <!-- JavaScript Bundle with Popper -->

  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/js/bootstrap.bundle.min.js"

    integrity="sha384-pprn3073KE6tl6bjs2QrFaJGz5/SUsLqktiwsUTF55Jfv3qYSDhgCecCxMW52nD2"

    crossorigin="anonymous"></script>

</head>

<body>

  <app-root></app-root>

</body>

</html>

Esta página es el entry point de la aplicación Angular. Tiene unas pocas modificaciones. Le puse el tag <title>, para identificar la pestaña en el browser. Notar que el favicon.ico no está en el directorio raíz, sino en la carpeta assets. Lo podemos poner en la carpeta que más nos convenga. Este es el modo de hacerlo. Agregué la hoja de estilo y la biblioteca de Bootstrap. Esta página será responsiva, es decir que se adaptará automáticamente a la pantalla en que la miremos. Finalmente, vemos que lo único que hay en el body es el tag <app-root></app-root>, que muestra el componente raíz de la aplicación Angular. Recordemos que en una aplicación Angular cada cosa que hay es un componente, y la aplicación es un conjunto de componentes anidados en otros componentes.

# src\main.ts

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

/\*

La clase AppModule tiene toda la información

sobre los otros componentes de la aplicación.

Está implementada, como se indica en el argumento

de from, en el archivo app.module.ts que está

en la carpeta ./app.

\*/

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.error(err));

Los otros archivos que están en el directorio src de la aplicación no los cambié, están tal como fueron generados.

# src\app\app.module.ts

import { HttpClientModule } from '@angular/common/http';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

/\*

Estas son las clases que exportan todos los componentes

que hemos creado nosotros para nuestra aplicación.

\*/

import { AppComponent } from './app.component';

import { AddComponent } from './components/add/add.component';

import { EditComponent } from './components/edit/edit.component';

import { HomeComponent } from './components/home/home.component';

@NgModule({

  declarations: [

    AppComponent,

    HomeComponent,

    EditComponent,

    AddComponent,

  ],

  imports: [

    AppRoutingModule,

    BrowserModule,

    FormsModule,

    HttpClientModule,

  ],

  providers: [],

  bootstrap: [AppComponent]

})

/\*

Esta es la clase del módulo principal de nuestra

aplicación, es la que hace que todo lo demás

funcione.

\*/

export class AppModule { }

# src\app\app-routing.module.ts

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

/\*

Necesitamos importar los componentes,

para luego poder usarlos. En esta aplicación tenemos

tres componentes, cada uno de los cuales está

implementado en 4 archivos, que están dentro de

una carpeta que lleva el nombre del componente

El argumento entre las llaves {} es el nombre de

la clase que el componente exporta.

El argumento from da el path relativo de la carpeta

en la que está el componente, y da también los dos

primeros tokens del nombre de los archivos. Omite

la extensión o las extensiones, porque son obvias.

Como resultado de la importación, este módulo en

el que ahora estamos, "conoce" o dispone de información

sobre cada uno de los componentes importados: la clase

que define su estado y comportamiento, el template que

esquematiza su contenido, la hoja de estilo que modifica

su aspecto, y eventualmente los tests que le haremos.

\*/

import { AddComponent } from './components/add/add.component';

import { EditComponent } from './components/edit/edit.component';

import { HomeComponent } from './components/home/home.component';

/\*

La constante routes, de tipo Routes, es un array de

objetos. Cada uno de esos obejtos define un end point.

El valor de la clave path es, naturalmente, lo que está

en la URL, además del nombre de dominio y el puerto.

Por ejemplo, en mi PC, la URL http://localhost:4200/home

hace que se muestre el componente Home, con la lista

de las tareas que haya en la base de datos.

\*/

const routes: Routes = [

  { path: '', redirectTo: 'home', pathMatch: 'full' },

  { path: 'home', component: HomeComponent },

  { path: 'edit/:id', component: EditComponent },

  { path: 'add', component: AddComponent }

];

/\*

https://angular.io/api/router/RouterModule

RouterModule

Agrega directivas y proveedores para la navegación

dentro de la aplicación entre las vistas

definidas en una aplicación. Utilice el

servicio de enrutador angular para especificar

de forma declarativa los estados de la

aplicación y administrar las transiciones

de estado.

El método forRoot() crea un NgModule

que contiene todas las directivas,

las rutas dadas y el propio servicio

de enrutador.

\*/

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

/\*

Esta es la clase que se exporta, para poder

ser usada en otros componentes.

\*/

export class AppRoutingModule { }

# src\app\app.component.ts

import { Component } from '@angular/core';

/\*

Estamos en el componente raíz de la aplicación.

Acá definimos el selector, es decir lo que va en el

tag que pretendemos usar para insertarlo.

Tanbién determinamos el template, ya sea porque damos

el nombre del archivo que lo contiene, como en este

caso, o porque lo damos inline.

Finalmente, determinamos la hoja de esilo usada.

\*/

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.scss']

})

/\*

Esta es la clase que exportamos, y es la que

tenemos que importar desde el archivo en el

que queramos incluir este componente.

Esta clase tiene una propiedad, que se llama

title, y contiene una string. Después veremos

que se usa con el mecamismo de interpolación,

en el template de este compoente.

\*/

export class AppComponent {

  title = 'Administrador de tareas';

}

# src\app\app.component.html

<!--

Interpolación de variables. Lo que está entre

las dobles llaves {{}} es la variable title,

cuyo valor se establece en la clase, y acá

es interpolado.

-->

<h1>{{title}}</h1>

<hr>

<!--

https://angular.io/api/router/RouterOutlet

RouterOutlet DIRECTIVE

Actúa como un marcador de posición que Angular

dinámicamente llena según el estado actual

del enrutador.

El <router-outlet> le dice al enrutador dónde

mostrar las vistas enrutadas. El RouterOutlet

es una de las directivas de enrutador

disponibles para AppComponent porque

AppModule importa AppRoutingModule que a su vez

exporta RouterModule.

-->

<router-outlet></router-outlet>

# Dockerizar MySQL

# 30-crud-spring-boot-angular-mysql-docker\compose.yaml

Creamos un archivo compose.yaml en la carpeta 30-crud-spring-boot-angular-mysql-docker.