Spring Boot con JPA

Gustavo García, 23/03/2023.

# Resumen

Trabajé sobre el artículo <https://spring.io/guides/gs/accessing-data-jpa/>

Este es un proyecto muy simple. Tiene la clase principal de la aplicación, una interfaz y la clase que modela la entidad. Usa la base de datos H2.

# Anotaciones de JPA

JPA (Java Persistence API) es una especificación de Java que define cómo se pueden persistir objetos Java en una base de datos relacional. Para ello, JPA utiliza un conjunto de anotaciones que se aplican a las clases y sus atributos. Algunas de las anotaciones más importantes son:

* @Entity: Se aplica a una clase que representa una entidad en la base de datos.
* @Id: Se aplica a un atributo que representa la clave primaria de la entidad.
* @GeneratedValue: Se utiliza en combinación con @Id para especificar cómo se debe generar la clave primaria (por ejemplo, mediante autoincremento).
* @Column: Se utiliza para especificar el nombre de la columna en la tabla que se corresponde con el atributo.
* @OneToMany y @ManyToOne: Se utilizan para especificar las relaciones entre entidades.
* @Transient: Se aplica a un atributo que no se debe persistir en la base de datos.

Estas son solo algunas de las anotaciones que se pueden utilizar en JPA. Las anotaciones permiten a los desarrolladores especificar cómo se deben mapear las clases y sus atributos a la base de datos, lo que facilita la persistencia de objetos Java.

# Crear nuevo proyecto de Java

Ctrl + shift + p

Java: Create Java Project…

Spring Boot

Maven Project

3.0.5

Java

tup

demo

Jar

17

# Dependencias

Spring Data JPA

H2 Database

# Clase Customer

Crear un paquete entities. En ese paquete pegar la siguiente clase:

package tup.demo.entities;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class Customer {

  @Id

  @GeneratedValue(strategy = GenerationType.AUTO)

  private Long id;

  private String firstName;

  private String lastName;

  protected Customer() {

  }

  public Customer(String firstName, String lastName) {

    this.firstName = firstName;

    this.lastName = lastName;

  }

  @Override

  public String toString() {

    return String.format(

        "Customer[id=%d, firstName='%s', lastName='%s']",

        id, firstName, lastName);

  }

  public Long getId() {

    return id;

  }

  public String getFirstName() {

    return firstName;

  }

  public String getLastName() {

    return lastName;

  }

}

La clase Customer tiene tres atributos: id, nombre y apellido. También tiene dos constructores. El constructor predeterminado existe solo por necesidad de JPA. No se usa directamente, por lo que se designa como protegido. El otro constructor es el que se usa para crear instancias de Customer para guardarlas en la base de datos.

La clase Customer se anota con @Entity, lo que indica que es una entidad JPA. (Debido a que no pusimos la anotación @Table, se supone que esta entidad está asignada a una tabla denominada Cliente).

La propiedad id del objeto Cliente se anota con @Id para que JPA la reconozca como la ID del objeto. La propiedad id también se anota con @GeneratedValue para indicar que la ID debe generarse automáticamente.

Las otras dos propiedades, firstName y lastName, se dejan sin anotar. Se supone que se asignan a columnas que tienen los mismos nombres que las propiedades.

El método toString() imprime las propiedades del cliente.

# Interfaz CustomerRepository

Crear un paquete repositories, y el él crear la interfaz siguiente:

package tup.demo.repositories;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import tup.demo.entities.Customer;

public interface CustomerRepository extends CrudRepository<Customer, Long> {

  List<Customer> findByLastName(String lastName);

  Customer findById(long id);

}

CustomerRepository extiende la interfaz CrudRepository. El tipo de entidad y el ID con el que trabaja, Customer y Long, se especifican en los parámetros genéricos en CrudRepository. Al extender CrudRepository, CustomerRepository hereda varios métodos para trabajar con la persistencia de Customer, incluidos métodos para guardar, eliminar y encontrar entidades de Customer.

Spring Data JPA también le permite definir otros métodos de consulta declarando su firma de método. Por ejemplo, CustomerRepository incluye el método findByLastName().

En una aplicación típica de Java, podría esperar escribir una clase que implemente CustomerRepository. Sin embargo, eso es lo que hace que Spring Data JPA sea tan poderoso: no necesita escribir una implementación de la interfaz del repositorio. Spring Data JPA crea una implementación cuando ejecuta la aplicación.

¡Ahora puede conectar este ejemplo y ver cómo se ve!

# CommandLineRunner

CommandLineRunner es una interfaz de Spring Boot que se utiliza para ejecutar código justo después de que se haya inicializado el contexto de la aplicación. Esta interfaz tiene un método run() que se invoca automáticamente después de que la aplicación se haya iniciado por completo.

El método run() de CommandLineRunner se utiliza comúnmente para realizar tareas de inicialización o para realizar algún tipo de proceso al inicio de la aplicación. Por ejemplo, se puede utilizar para cargar datos iniciales en una base de datos, para configurar objetos de la aplicación, para establecer propiedades del sistema, para inicializar el estado de la aplicación, entre otras cosas.

Para utilizar CommandLineRunner en una aplicación Spring Boot, se debe implementar la interfaz y sobrescribir el método run(). Luego, se debe agregar el bean a la clase principal de la aplicación usando la anotación @Component o alguna otra anotación relacionada, para que la aplicación detecte y ejecute el código en el momento adecuado.

En resumen, CommandLineRunner es una interfaz de Spring Boot que permite ejecutar código automáticamente después de que se haya iniciado completamente la aplicación, y se utiliza comúnmente para realizar tareas de inicialización y configuración.

# Anotación @Bean

La anotación @Bean es una anotación de Spring Framework que se utiliza para indicar que un método de una clase Java es un método factoría, es decir, un método que devuelve un objeto que Spring debe administrar como un componente dentro del contenedor de la aplicación.

Cuando se utiliza la anotación @Bean en un método, Spring Framework detecta automáticamente el método y se encarga de llamarlo para obtener el objeto y agregarlo al contexto de la aplicación como un componente de Spring. Los métodos anotados con @Bean pueden tener cualquier nombre, y el objeto que se devuelve puede ser de cualquier tipo.

La anotación @Bean se utiliza comúnmente para configurar y personalizar objetos que Spring debe administrar, como servicios, controladores, repositorios, componentes, entre otros. También se puede utilizar para crear objetos personalizados que se deben agregar al contexto de Spring y que se utilizarán en otras partes de la aplicación.

En resumen, la anotación @Bean es una anotación de Spring Framework que se utiliza para indicar que un método de una clase Java es un método de fábrica de objetos, que Spring Framework utilizará para crear y administrar objetos dentro del contexto de la aplicación.

La versión en inglés es mucho más clara.

The @Bean annotation in Spring Framework is used to indicate that a method in a Java class is a factory method that creates and returns a Spring-managed bean instance.

When the @Bean annotation is used on a method, Spring Framework automatically detects the method and calls it to obtain the object instance, which is then managed by Spring as a component within the application context. Methods annotated with @Bean can have any name, and the object instance returned by the method can be of any type.

The @Bean annotation is commonly used to configure and customize Spring-managed objects such as services, controllers, repositories, and other components. It can also be used to create custom objects that need to be added to the Spring context and used in other parts of the application.

In summary, the @Bean annotation in Spring Framework is used to mark a method as a factory method for creating and returning a Spring-managed object instance, which is then managed by Spring within the application context.

# Clase DemoApplication

Editar la clase DemoApplication para que quede así:

package tup.demo;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import tup.demo.entities.Customer;

import tup.demo.repositories.CustomerRepository;

@SpringBootApplication

public class DemoApplication {

    private static final Logger log = LoggerFactory.getLogger(DemoApplication.class);

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

    @Bean

    public CommandLineRunner demo(CustomerRepository repository) {

        return (args) -> {

            // save a few customers

            repository.save(new Customer("Jack", "Bauer"));

            repository.save(new Customer("Chloe", "O'Brian"));

            repository.save(new Customer("Kim", "Bauer"));

            repository.save(new Customer("David", "Palmer"));

            repository.save(new Customer("Michelle", "Dessler"));

            // fetch all customers

            log.info("Customers found with findAll():");

            log.info("-------------------------------");

            for (Customer customer : repository.findAll()) {

                log.info(customer.toString());

            }

            log.info("");

            // fetch an individual customer by ID

            Customer customer = repository.findById(1L);

            log.info("Customer found with findById(1L):");

            log.info("--------------------------------");

            log.info(customer.toString());

            log.info("");

            // fetch customers by last name

            log.info("Customer found with findByLastName('Bauer'):");

            log.info("--------------------------------------------");

            repository.findByLastName("Bauer").forEach(bauer -> {

                log.info(bauer.toString());

            });

            // for (Customer bauer : repository.findByLastName("Bauer")) {

            // log.info(bauer.toString());

            // }

            log.info("");

        };

    }

}

La clase DemoApplication incluye un método demo() que somete a CustomerRepository a algunas pruebas. Primero, obtiene CustomerRepository del contexto de la aplicación Spring. Luego guarda un puñado de objetos Customer, demostrando el método save() y configurando algunos datos para trabajar. A continuación, llama a findAll() para obtener todos los objetos Customer de la base de datos. Luego llama a findById() para obtener un solo Customer por su ID. Finalmente, llama a findByLastName() para encontrar todos los clientes cuyo apellido es "Bauer". El método demo() devuelve un bean CommandLineRunner que ejecuta automáticamente el código cuando se inicia la aplicación.

De forma predeterminada, Spring Boot habilita la compatibilidad con el repositorio JPA y busca en el paquete (y sus subpaquetes) donde se encuentra @SpringBootApplication. Si su configuración tiene definiciones de interfaz de repositorio JPA ubicadas en un paquete que no está visible, puede señalar paquetes alternativos utilizando @EnableJpaRepositories y su parámetro de tipo seguro basePackageClasses=MyRepository.class.

# Probar

Para compilar el proyecto se puede escribir en la terminal:

mvn clean install

Se puede ejecutar la aplicación apretando ctrl + F5.

Alternativamente, se puede ejecutar desde la terminal con:

java -jar target/demo-0.0.1-SNAPSHOT.jar

La salida de la aplicación es:

== Customers found with findAll():

Customer[id=1, firstName='Jack', lastName='Bauer']

Customer[id=2, firstName='Chloe', lastName='O'Brian']

Customer[id=3, firstName='Kim', lastName='Bauer']

Customer[id=4, firstName='David', lastName='Palmer']

Customer[id=5, firstName='Michelle', lastName='Dessler']

== Customer found with findById(1L):

Customer[id=1, firstName='Jack', lastName='Bauer']

== Customer found with findByLastName('Bauer'):

Customer[id=1, firstName='Jack', lastName='Bauer']

Customer[id=3, firstName='Kim', lastName='Bauer']

Cuidado, porque a la izquierda de cada una de estas líneas puede haber más texto, generado automáticamente por el logger.