Spring Boot ITS

RESTful Web Services và Tích hợp Cơ sở Dữ liệu

# Tổng quan về Spring Boot và cấu trúc dự án

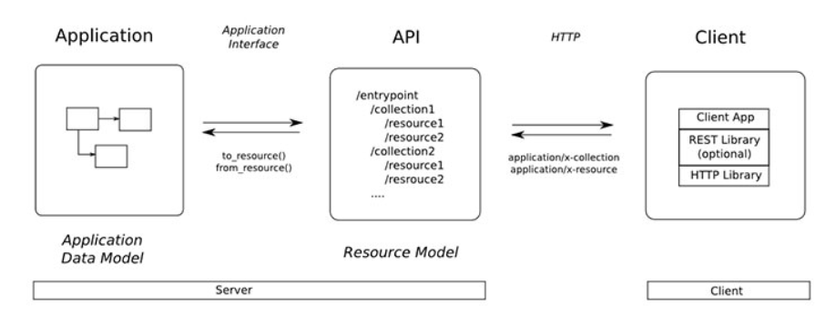
1. Xây dựng RESTful API với Spring Boot: GET, POST
2. RESTful API

* API (Application Programming Interface) là một tập các quy tắc và cơ chế mà theo đó, một ứng dụng hay một thành phần sẽ tương tác với một ứng dụng hay thành phần khác. API có thể trả về dữ liệu mà bạn cần cho ứng dụng của mình ở những kiểu dữ liệu phổ biến như JSON hay XML.
* RESTful API là một loại dịch vụ web dựa trên kiến trúc REST (Representational State Transfer). Nó cho phép các ứng dụng giao tiếp với nhau thông qua các yêu cầu HTTP, sử dụng các phương thức như GET, POST, PUT, DELETE.

1. phương thức HTTP

* GET (SELECT): Trả về một Resource hoặc một danh sách Resource.
* POST (CREATE): Tạo mới một Resource.
* PUT (UPDATE): Cập nhật thông tin cho Resource.
* DELETE (DELETE): Xoá một Resource.

1. RESTful API hoạt động như thế nào



* RESTful API hoạt động bằng cách sử dụng các phương thức HTTP để thực hiện các hành động trên tài nguyên. Dưới đây là quy trình cơ bản của một yêu cầu đến RESTful API:

1. **Client** gửi một yêu cầu HTTP (HTTP request) đến một URI (đại diện cho tài nguyên) với một phương thức HTTP nhất định.
2. **Server** nhận yêu cầu, xử lý dựa trên phương thức HTTP (GET, POST, PUT, DELETE), và thực hiện hành động tương ứng trên tài nguyên.
3. **Server** trả về phản hồi với dữ liệu đại diện cho tài nguyên (nếu có), cùng với mã trạng thái HTTP cho biết kết quả của yêu cầu.
4. Authentication request và cấu trúc dữ liệu trả về

* RESTful API không sử dụng session và cookie, nó sử dụng một access\_token với mỗi request. Tìm hiểu JWT (JsonWebToken) ở phần sau :v. Dữ liệu trả về thường có cấu trúc json như :

{

"status\_code": 200,

"data": [

{

"name": "ManhLD",

"email": "manhld@example.com",

"ny": "not found"

},

{

"name": "Ahri",

"email": "ahriKDA@lmht.com",

"ny": "Ezreal"

}

],

error\_messages: ""

}

1. Một số status code hay gặp

* **200 OK**: Yêu cầu đã được xử lý thành công (dành cho GET, PUT).
* **201 Created**: Tài nguyên đã được tạo thành công (dành cho POST).
* **204 No Content**: Yêu cầu thành công nhưng không có nội dung trả về (thường dùng cho DELETE).
* **400 Bad Request**: Yêu cầu không hợp lệ (thường là dữ liệu đầu vào không đúng).
* **401 Unauthorized**: Yêu cầu cần xác thực, nhưng thông tin xác thực không chính xác hoặc không tồn tại.
* **403 Forbidden**: Người dùng không có quyền truy cập tài nguyên.
* **404 Not Found**: Tài nguyên không tồn tại.
* **500 Internal Server Error**: Có lỗi xảy ra trên server.

1. Tạo REST controller

* **REST Controller** là nơi bạn định nghĩa các API endpoint để client có thể tương tác với hệ thống của bạn thông qua HTTP. Khi tạo **REST Controller** trong Spring Boot là đang tạo ra các API endpoint để client có thể giao tiếp với server thông qua các yêu cầu HTTP (GET, POST, PUT, DELETE).

1. Tạo restfullapi với get, post cho user

* Khi tạo một **controller** trong Spring Boot tức đang xây dựng các điểm cuối (endpoints) của một **RESTful API**. Một **controller** chịu trách nhiệm xử lý các yêu cầu HTTP (GET, POST, PUT, DELETE,...) và trả về dữ liệu hoặc phản hồi từ server. Controller đã được tìm hiểu trong task trước nên task này em chỉ bổ sung kiến thức mới học được
* Trước tiên tạo class user để mục đích sau tạo 1 list user và fetch dữ liệu từ file json, get thêm sửa xoá (được định nghĩa trong service) với list đó. tạo api với controller để trả ra dữ liệu json
* Tạo class user
* Khai báo các thuộc tính, constructer, getter, setter :v
* Tạo user service
* fech user từ json vào list user

@PostConstruct

private void fetchUsersFromJson() {

ObjectMapper mapper = new ObjectMapper();

TypeReference<List<User>> typeReference = new TypeReference<>() {};

InputStream inputStream = TypeReference.class.getResourceAsStream("/data.json");

try {

if (inputStream != null) {

users = mapper.readValue(inputStream, typeReference);

} else {

throw new IllegalArgumentException("data.json file not found!");

}

} catch (Exception e) {

logger.error("Error loading users", e);

}

}

* read all obj in json

public List<User> getAll() { return users; }

* trả về 1 record với id phù hợp

public Optional<User> findById(Long id) {

return users.stream().filter(user -> user.getId().equals(id)).findFirst();

}

* create

public User addUser(User user) {

users.add(user);

return user;

}

* update

public User update(Long id, User updatedUser) {

Optional<User> userOptional = findById(id);

if (userOptional.isPresent()) {

User user = userOptional.get();

user.setName(updatedUser.getName());

user.setEmail(updatedUser.getEmail());

return user;

}

return null;

}

* delete

public void deleteById(Long id) {

users.removeIf(user -> user.getId().equals(id));

}

* loading ♾️
* Tạo user controller
* constructer ( giải thích kỹ hơn ở mục 8)

// dependency injection ví dụ thực tế đây

public UserController(UserService userService) {

this.userService = userService;

}

* hàm lấy ra tất cả user (get)

@GetMapping

public List<User> getAllUsers() {

return userService.getAll();

}

* hàm lấy ra user theo id (get)

@GetMapping("/{id}")

public ResponseEntity<User> getUserById(@PathVariable Long id) {

Optional<User> user = userService.findById(id);

return user.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

* hàm tạo mới user (post)

@PostMapping("/add")

public User createUser(@RequestBody User user) {

return userService.addUser(user);

}

* hàm update info (post)

@PostMapping("/update/{id}")

public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User updatedUser) {

User user = userService.update(id, updatedUser);

return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();

}

* hàm xoá user (get)

@GetMapping("/delete/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable Long id) {

userService.deleteById(id);

return ResponseEntity.noContent().build();

}

* loading ♾️
* loading ♾️

## Các kiến thức mới qua xây dựng api, các kiến thức liên quan từ task trước được làm rõ qua code thực hiện api

1. ObjectMapper

* ObjectMapper dùng để **chuyển đổi giữa các đối tượng Java và dữ liệu JSON**. Đây là một trong những công cụ mạnh mẽ và phổ biến nhất trong Jackson để thực hiện việc serialize (chuyển đối tượng Java thành chuỗi JSON) và deserialize (chuyển chuỗi JSON thành đối tượng Java).
* Các chức năng chính của ObjectMapper
* Serialization: Chuyển đối tượng Java thành JSON.
* Deserialization: Chuyển chuỗi JSON thành đối tượng Java.
* Đọc và ghi tệp JSON: Có thể đọc và ghi JSON từ tệp, luồng, hoặc chuỗi.
* Cấu hình và tùy chỉnh: Cho phép cấu hình để xử lý các đặc điểm của đối tượng JSON và Java.
* Serialization (chuyển đối tượng Java thành JSON)

json = objMapper.writeValueAsString(obj)

import com.fasterxml.jackson.databind.ObjectMapper;

public class Main {

public static void main(String[] args) {

ObjectMapper objectMapper = new ObjectMapper();

// Tạo đối tượng Java

User user = new User("John", 25);

try {

// Chuyển đổi đối tượng Java thành chuỗi JSON

String jsonString = objectMapper.writeValueAsString(user);

System.out.println(jsonString);

} catch (Exception e) {

e.printStackTrace();

}

}

}

class User {

private String name;

private int age;

public User() {}

public User(String name, int age) {

this.name = name;

this.age = age;

}

// Getters và setters

}

* Deserialization (chuyển JSON thành đối tượng Java)

|  |  |
| --- | --- |
| Phương thức | Mô tả |
| T readValue(String content, Class<T> valueType) | Đọc JSON từ chuỗi. |
| T readValue(File src, Class<T> valueType) | Đọc JSON từ tệp. |
| T readValue(InputStream src, Class<T> valueType) | Đọc JSON từ InputStream. |
| T readValue(URL src, Class<T> valueType) | Đọc JSON từ URL. |
| T readValue(String content, TypeReference<T> valueTypeRef) | Đọc JSON từ chuỗi với TypeReference cho kiểu dữ liệu phức tạp. |

objString = objMapper.readValue(json, obj.class)

import com.fasterxml.jackson.databind.ObjectMapper;

public class Main {

public static void main(String[] args) {

ObjectMapper objectMapper = new ObjectMapper();

// Chuỗi JSON

String jsonString = "{\"name\":\"John\", \"age\":25}";

try {

// Chuyển đổi chuỗi JSON thành đối tượng Java

User user = objectMapper.readValue(jsonString, User.class);

System.out.println(user.getName() + " - " + user.getAge());

} catch (Exception e) {

e.printStackTrace();

}

}

}

class User {

private String name;

private int age;

public User() {}

public User(String name, int age) {

this.name = name;

this.age = age;

}

// Getters và setters

}

* Đọc JSON từ tệp

private void fetchUsersFromJson() {

ObjectMapper mapper = new ObjectMapper();

TypeReference<List<User>> typeReference = new TypeReference<>() {};

InputStream inputStream = TypeReference.class.getResourceAsStream("/data.json");

try {

if (inputStream != null) {

users = mapper.readValue(inputStream, typeReference);

} else {

throw new IllegalArgumentException("data.json file not found!");

}

} catch (Exception e) {

logger.error("Error loading users", e);

}

}

* Ghi JSON vào tệp

import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.File;

public class Main {

public static void main(String[] args) {

ObjectMapper objectMapper = new ObjectMapper();

User user = new User("John", 25);

try {

// Ghi JSON vào tệp

objectMapper.writeValue(new File("user.json"), user);

} catch (Exception e) {

e.printStackTrace();

}

}

}

* Tùy chỉnh cấu hình
* Ví dụ: Nếu JSON không chứa tất cả các trường của đối tượng Java, bạn có thể cho phép Jackson bỏ qua các trường thiếu bằng cách cấu hình ObjectMapper:

ObjectMapper objectMapper = new ObjectMapper();

objectMapper.configure(DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES, false);

* TypeReference cho generic types
* Khi deserialize dữ liệu JSON thành các kiểu dữ liệu phức tạp, như List<User>, bạn cần sử dụng TypeReference để Jackson biết chính xác loại dữ liệu bạn mong muốn.

ObjectMapper objectMapper = new ObjectMapper();

String jsonString = "[{\"name\":\"John\", \"age\":25}, {\"name\":\"Doe\", \"age\":30}]";

try {

// Sử dụng TypeReference để deserialize JSON thành List<User>

List<User> users = objectMapper.readValue(jsonString, new TypeReference<List<User>>() {});

users.forEach(user -> System.out.println(user.getName() + " - " + user.getAge()));

} catch (Exception e) {

e.printStackTrace();

}

* loading ♾️

1. TypeReference

* TypeReference là một lớp trong thư viện Jackson được sử dụng để chỉ rõ kiểu dữ liệu phức tạp (generics) khi làm việc với phương thức readValue() của ObjectMapper. Điều này đặc biệt cần thiết khi bạn muốn deserialize (chuyển đổi) JSON thành một danh sách, tập hợp, hoặc một loại dữ liệu tổng quát nào đó như List<User>, Map<String, User>, v.v.
* Java sử dụng khái niệm **type erasure** cho các loại dữ liệu tổng quát (generics). Điều này có nghĩa là tại thời điểm runtime, thông tin về các kiểu generic như List<User> sẽ bị mất đi, chỉ còn lại thông tin chung chung như List. Do đó, khi Jackson cố gắng deserialize JSON thành một List<User>, nó không thể xác định được kiểu User bên trong danh sách, dẫn đến lỗi. TypeReference được sử dụng để cung cấp thông tin về kiểu dữ liệu cụ thể mà Jackson cần trong quá trình deserialize.
* Deserialize JSON thành List<User>

public class Main {

public static void main(String[] args) {

ObjectMapper objectMapper = new ObjectMapper();

String jsonArray = "[{\"name\":\"John\", \"age\":30}, {\"name\":\"Jane\", \"age\":25}]";

try {

// Sử dụng TypeReference để chỉ rõ kiểu List<User>

List<User> users = objectMapper.readValue(jsonArray, new TypeReference<List<User>>() {});

users.forEach(user -> System.out.println(user.getName() + " - " + user.getAge()));

} catch (Exception e) {

e.printStackTrace();

}

}

}

* Deserialize JSON thành Map<String, User>

String jsonMap = "{\"user1\": {\"name\": \"John\", \"age\": 30}, \"user2\": {\"name\": \"Jane\", \"age\": 25}}";

Map<String, User> userMap = objectMapper.readValue(jsonMap, new TypeReference<Map<String, User>>() {});

userMap.forEach((key, user) -> System.out.println(key + ": " + user.getName() + " - " + user.getAge()));

* loading ♾️

1. Rõ hơn DI, IoC với constructor bên trên

* khi chương trình khởi động, Spring sẽ tự động quét các lớp được chú thích bằng các annotation như @RestController, @Service, @Component, @Repository, v.v.
* Spring tìm thấy lớp UserController (được chú thích với @RestController hoặc @Controller), nó sẽ tự động tạo ra instance của UserController. Khi đó, hàm tạo của lớp này sẽ được gọi.
* UserService được chú thích bằng @Service, nghĩa là nó cũng là một Bean do Spring quản lý.
* Khi Spring tạo ra UserController, nó kiểm tra các dependency của lớp này. Thấy rằng UserController cần một UserService, Spring sẽ tìm một Bean UserService đã được tạo sẵn và truyền nó vào hàm tạo của UserController

1. cố lên :<
2. Tích hợp cơ sở dữ liệu với JPA/Hibernate: Entity, Repository, Service. + Cấu hình kết nối cơ sở dữ liệu MySQL (Đa số các dự án làm việc với MySQL. Có thể tìm hiểu thêm PostgreSQL, mongoDb, redis,....). + CRUD (Create, Read, Update, Delete) với cơ sở dữ liệu.
3. Giới thiệu về JPA và Hibernate trong Spring Boot

* **JPA (Java Persistence API)** là một chuẩn để quản lý dữ liệu quan hệ trong các ứng dụng Java. Nó giúp ánh xạ các đối tượng Java với các bảng trong cơ sở dữ liệu quan hệ mà không cần viết các câu lệnh SQL phức tạp.
* **Hibernate** là một framework ORM phổ biến nhất trong cộng đồng Java, và nó là một trong những triển khai cụ thể của JPA. Hibernate cung cấp nhiều tính năng mở rộng mà JPA không có, như quản lý bộ nhớ đệm (caching), lazy loading, và hỗ trợ native SQL.
* loading ♾️

1. Entity, Repository, và Service

* **Entity**: Đây là các lớp Java biểu diễn các bảng trong cơ sở dữ liệu. Mỗi thuộc tính của lớp sẽ ánh xạ đến một cột trong bảng.
* **Repository**: Đây là nơi chứa các thao tác CRUD (Create, Read, Update, Delete) cho các entity. Spring Data JPA cung cấp các phương thức có sẵn để tương tác với cơ sở dữ liệu và cũng có thể tự định nghĩa thêm.
* **Service**: Service quản lý logic nghiệp vụ của ứng dụng, xử lý các thao tác CRUD bằng cách gọi repository và có thể chứa thêm các logic nghiệp vụ khác.
* loading ♾️

1. Cách kết nối spring boot với DB

* Cấu hình kết nối
* Trong build.gradle thêm dependency

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

implementation 'org.mariadb.jdbc:mariadb-java-client:3.1.2

' // For MariaDB

// or use MySQL

implementation 'mysql:mysql-connector-java:8.0.32

'

* Tuỳ chỉnh cấu hình với application.properties

spring.datasource.url=jdbc:mariadb://localhost:3306/db\_name

#nhưng e cài qua docker nên set port 3307 chứ 3306 trùng mariadb rồi :)

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

#spring.jpa.hibernate.ddl-auto=create #tạo mới table & dữ liệu khi build

spring.jpa.hibernate.ddl-auto=update #ko tạo mới table, dữ liệu vẫn còn trong DB

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

* loading ♾️
* loading ♾️

1. Các annotation cho entity

* @Entity
* Đánh dấu một lớp là một **thực thể** (entity) và ánh xạ với một bảng trong cơ sở dữ liệu.
* Ví dụ:

@Entity

public class Book {

// fields and methods

}

* loading ♾️
* @Table
* Dùng để chỉ định tên bảng mà thực thể được ánh xạ đến. Nếu không dùng annotation này, JPA sẽ mặc định dùng tên lớp làm tên bảng.
* Thuộc tính quan trọng:
* name: Tên bảng trong cơ sở dữ liệu.
* Ví dụ:

@Entity

@Table(name = "books")

public class Book {

// fields and methods

}

* loading ♾️
* loading ♾️

1. Annotation cho khóa chính

* **@Id**
* Chỉ định thuộc tính làm **khóa chính** của thực thể.
* Ví dụ:

@Id

private Long id;

* loading ♾️
* @GeneratedValue
* Chỉ định strategy tạo giá trị cho khóa chính.
* Thuộc tính:
* strategy: Cách tạo giá trị khóa chính (ví dụ: GenerationType.AUTO, IDENTITY, SEQUENCE, TABLE).
* nếu để sequence thì trong db nó tạo ra 1 table nữa có đuôi \_seq để lưu giá trị của id mới nhất được tạo, ko ấy dùng identity thôi
* Ví dụ

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

* loading ♾️
* **@SequenceGenerator**:
* Được sử dụng cùng với @GeneratedValue để tạo khóa chính dựa trên một **sequence** (chuỗi số tuần tự).
* Ví dụ:

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book\_seq")

@SequenceGenerator(name = "book\_seq", sequenceName = "book\_sequence", allocationSize = 1)

private Long id;

* loading ♾️
* loading ♾️

1. Annotation cho các trường

* **@Column**:
* Ánh xạ một thuộc tính của thực thể tới một cột trong bảng cơ sở dữ liệu.
* Thuộc tính quan trọng:
* name: Tên cột trong DB (ánh xạ đến thuộc tính tương ứng)
* nullable: Cột có thể để trống hay không.
* length: Độ dài của cột (cho kiểu chuỗi).
* unique: Cột có giá trị duy nhất không.
* loading ♾️
* Ví dụ

@Column(name = "book\_title", nullable = false, length = 100)

private String title;

* loading ♾️
* @JoinColumn:
* Kiểu như khoá ngoại với bảng khác :v
* Ví dụ

@OneToOne

@JoinColumn(name = "author\_id")

private Author author;

* **@Transient**:
* Chỉ định một thuộc tính **không** được lưu trữ trong cơ sở dữ liệu.
* Ví dụ:

@Transient

private String tempValue;

* **@Temporal**:
* Chỉ định kiểu dữ liệu ngày giờ cho các trường Date hoặc Calendar.
* Thuộc tính quan trọng:
* TemporalType.DATE: Chỉ lưu ngày (yyyy-MM-dd).
* TemporalType.TIME: Chỉ lưu thời gian (HH:mm  
  ).
* TemporalType.TIMESTAMP: Lưu cả ngày và giờ.
* loading ♾️
* Ví dụ:

@Temporal(TemporalType.DATE)

private Date publishedDate;

* loading ♾️
* **@Lob**:
* Dùng để chỉ định các trường kiểu dữ liệu lớn như **BLOB** (Binary Large Object) hoặc **CLOB** (Character Large Object).
* Ví dụ:

@Lob

private byte[] content;

* loading ♾️
* **@Enumerated**:
* Dùng để ánh xạ một trường kiểu enum sang cột trong cơ sở dữ liệu.
* Thuộc tính quan trọng:
* EnumType.ORDINAL: Lưu chỉ số của enum (0, 1, 2,...).
* EnumType.STRING: Lưu tên của enum dưới dạng chuỗi.
* loading ♾️
* Ví dụ:
* Book với **@Enumerated**

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String title;

@Enumerated(EnumType.STRING) // Lưu enum dưới dạng chuỗi

@Column(name = "book\_status")

private BookStatus status;

// Constructors, Getters, Setters

}

* Enum BookStatus

public enum BookStatus {

AVAILABLE,

OUT\_OF\_STOCK,

DISCONTINUED

}

* Tạo bảng Book trong DB

CREATE TABLE Book (

id BIGINT NOT NULL AUTO\_INCREMENT,

title VARCHAR(255) NOT NULL,

book\_status VARCHAR(255),

PRIMARY KEY (id)

);

* Thử nghiệm API

{

"title": "Spring Boot in Action",

"status": "AVAILABLE"

}

* loading ♾️
* loading ♾️
* loading ♾️

1. Annotation cho các quan hệ giữa các Entity

* Hibernate cung cấp hai cơ chế lấy dữ liệu: Lazy Loading, Eager Loading
* **Lazy Loading**: Dữ liệu chỉ được lấy khi thực sự cần thiết. Ví dụ, nếu bạn không gọi đến phương thức getBooks(), dữ liệu sách sẽ không được tải.
* **Eager Loading**: Dữ liệu được lấy ngay lập tức, ngay cả khi không cần thiết.
* **@OneToOne**:
* Thuộc tính quan trọng:
* mappedBy: Xác định bên "chủ" của mối quan hệ và chỉ ra rằng khóa ngoại được quản lý bởi đối tượng khác. Dùng trong entity phụ thuộc (inverse side) của mối quan hệ.
* fetch: Chiến lược lấy dữ liệu (mặc định là FetchType.EAGER).
* cascade: xác định hành vi của các thao tác liên quan đến các thực thể liên kết. Khi một thực thể chính thực hiện thao tác (lưu, cập nhật, xóa), các thao tác này có thể được truyền cho các thực thể liên quan thông qua **Cascade**.
* optional: Cho biết đối tượng liên quan có thể là null hay không. Nếu false, thì đối tượng liên quan bắt buộc phải tồn tại. Mặc định là true
* orphanRemoval: Nếu được đặt là true, các đối tượng liên quan không còn được tham chiếu sẽ tự động bị xóa khỏi cơ sở dữ liệu.
* Ví dụ, nếu một thực thể **Author** có quan hệ với **Book**, khi xóa **Author**, chúng ta có thể muốn xóa tất cả các cuốn sách liên quan. Điều này được thực hiện thông qua **Cascade**.
* Các loại **CascadeType**
* **ALL**: Bao gồm tất cả các loại thao tác.
* **PERSIST**: Khi lưu thực thể chính, các thực thể liên quan cũng được lưu.
* **MERGE**: Khi hợp nhất thực thể chính, các thực thể liên quan cũng được hợp nhất.
* **REMOVE**: Khi xóa thực thể chính, các thực thể liên quan cũng bị xóa.
* **REFRESH**: Khi làm mới thực thể chính, các thực thể liên quan cũng được làm mới.
* **DETACH**: Khi tách thực thể chính ra khỏi ngữ cảnh persistence, các thực thể liên quan cũng bị tách ra.
* loading ♾️
* loading ♾️
* Ví dụ:
* tạo author

@OneToOne

@JoinColumn(name = "author\_id")

private Author author;

@Entity

public class Author {

...

// tạo thuộc tính book trong author

@OneToOne(mappedBy = "author", cascade = CascadeType.ALL)

private Book book;

// Constructor chỉ cần truyền vào tên tác giả, có thể sử dụng phương thức setBook() để thiết lập mối quan hệ với Book sau khi cả hai đối tượng đã được khởi tạo.

public Author(String name) {

this.name = name;

}

// thêm getter, setter cho book vào

public Book getBook() {

return book;

}

public void setBook(Book book) {

this.book = book;

}

...

}

* tạo book

@Entity

public class Book {

...

// tạo thuộc tính kiểu author trong book

@OneToOne

@JoinColumn(name = "author\_id")

private Author author;

//constructer nhét thêm thằng author vào

public Book(String title, Author author) {

this.title = title;

this.author = author;

}

// thêm getter, setter cho author zô

public Author getAuthor() {

return author;

}

public void setAuthor(Author author) {

this.author = author;

}

...

}

* repo cho cả book và author

public interface BookRepository extends JpaRepository<Book, Long> {}

public interface AuthorRepository extends JpaRepository<Author, Long> {}

* Sử dụng trong Service

@Service

public class LibraryService {

@Autowired

private BookRepository bookRepository;

@Autowired

private AuthorRepository authorRepository;

@Transactional

public void createBookWithAuthor() {

Author author = new Author("J.K. Rowling");

Book book = new Book("Harry Potter", author);

// Thiết lập liên kết 2 chiều

author.setBook(book);

// Lưu tác giả và sách

authorRepository.save(author);

bookRepository.save(book);

}

}

* loading ♾️
* loading ♾️
* @OneToMany and @ManyToOne:
* Thuộc tính chính:
* mappedBy: Chỉ ra tên của trường tham chiếu bên phía entity nhiều (@ManyToOne).
* cascade: Tương tự như @OneToOne, xác định hành vi cascade từ entity chủ đến các entity con.
* fetch: Cách nạp dữ liệu (mặc định là FetchType.LAZY).
* orphanRemoval: Nếu true, các entity con không còn được tham chiếu sẽ tự động bị xóa.
* Ví dụ để hiểu
* tạo author

@Entity

public class Author {

...

@OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Book> books;

public Author(String name) {

this.name = name;

}

public List<Book> getBooks() {

return books;

}

public void setBooks(List<Book> books) {

this.books = books;

}

...

}

* tạo book

@Entity

public class Book {

...

@ManyToOne

@JoinColumn(name = "author\_id")

private Author author;

public Book(String title, Author author) {

this.title = title;

this.author = author;

}

public Author getAuthor() {

return author;

}

public void setAuthor(Author author) {

this.author = author;

}

...

}

* repo

@Repository

public interface AuthorRepository extends JpaRepository<Author, Long> {

// Bạn có thể thêm các phương thức tùy chỉnh tại đây nếu cần

}

@Repository

public interface BookRepository extends JpaRepository<Book, Long> {

// Bạn có thể thêm các phương thức tùy chỉnh tại đây nếu cần

}

* controller

@PostMapping("/{authorId}/books")

public Book addBookToAuthor(@PathVariable Long authorId, @RequestBody Book book) {

Author author = authorRepository.findById(authorId).orElse(null);

if (author != null) {

book.setAuthor(author); // Thiết lập tác giả cho sách

author.getBooks().add(book); // Thêm sách vào danh sách sách của tác giả

authorRepository.save(author); // Lưu tác giả (cùng với sách)

return bookRepository.save(book); // Lưu sách

}

return null;

}

getBooks() trả về list nên có phương thức add

* Giải thích
* Một tác giả (Author) có thể viết nhiều cuốn sách (Book). Trong trường hợp này, Author có mối quan hệ @OneToMany với Book.
* Nhiều cuốn sách (Book) có thể thuộc về một tác giả (Author). Trong trường hợp này, Book có mối quan hệ @ManyToOnevới Author.
* @OneToMany(mappedBy = "author"): Điều này chỉ ra rằng danh sách sách (books) trong Author là "được quản lý" bởi trường author trong entity Book
* @JoinColumn(name = "author\_id"): Điều này tạo cột author\_id trong bảng Book, dùng để lưu trữ khóa ngoại liên kết với Author.
* loading ♾️
* @ManyToMany:
* Thuộc tính chính:
* cascade: Xác định hành vi cascade giữa các đối tượng.
* fetch: Mặc định là FetchType.LAZY, nhưng có thể chuyển thành FetchType.EAGER.
* mappedBy: Chỉ ra tên của trường tham chiếu ở entity đối nghịch trong mối quan hệ.
* JoinTable: Được sử dụng để chỉ định bảng trung gian khi thiết lập mối quan hệ nhiều - nhiều, với các khóa ngoại từ cả hai bảng.
* ví dụ
* tạo student

@Entity

public class Student {

...

@ManyToMany(mappedBy = "students", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private Set<Course> courses = new HashSet<>();

public Student(String name) {

this.name = name;

}

public Set<Course> getCourses() {

return courses;

}

public void setCourses(Set<Course> courses) {

this.courses = courses;

}

...

}

* tạo course

@Entity

public class Course {

...

@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)

@JoinTable(

name = "student\_course", // Tên bảng liên kết

joinColumns = @JoinColumn(name = "course\_id"), // Khóa ngoại cho Course

inverseJoinColumns = @JoinColumn(name = "student\_id") // Khóa ngoại cho Student

)

private Set<Student> students = new HashSet<>();

public Course(String title) {

this.title = title;

}

public Set<Student> getStudents() {

return students;

}

public void setStudents(Set<Student> students) {

this.students = students;

}

...

}

* Repo

@Repository

public interface StudentRepository extends JpaRepository<Student, Long> {

// Bạn có thể thêm các phương thức tùy chỉnh tại đây nếu cần

}

@Repository

public interface CourseRepository extends JpaRepository<Course, Long> {

// Bạn có thể thêm các phương thức tùy chỉnh tại đây nếu cần

}

* sử dụng

@PostMapping("/{studentId}/courses")

public Course addCourseToStudent(@PathVariable Long studentId, @RequestBody Course course) {

Student student = studentRepository.findById(studentId).orElse(null);

if (student != null) {

course.getStudents().add(student); // Thêm sinh viên vào danh sách sinh viên của khóa học

student.getCourses().add(course); // Thêm khóa học vào danh sách khóa học của sinh viên

studentRepository.save(student); // Lưu sinh viên

return courseRepository.save(course); // Lưu khóa học

}

return null;

}

* loading ♾️
* loading ♾️
* loading ♾️

1. Annotation cho truy vấn

* **@Query**:
* Được dùng để định nghĩa truy vấn **JPQL** hoặc **Native SQL**.
* Thuộc tính quan trọng:
* nativeQuery: Chỉ định nếu truy vấn là SQL gốc.
* Ví dụ:

@Query("SELECT b FROM Book b WHERE b.author = :author")

List<Book> findBooksByAuthor(@Param("author") String author);

@Query(value = "SELECT \* FROM books WHERE price > :price", nativeQuery = true)

List<Book> findBooksByPriceGreaterThan(@Param("price") Double price);

* loading ♾️

1. Annotation cho Transaction Management (Quản lý giao dịch)

* **@Transactional**:
* Chỉ định rằng phương thức hoặc lớp được thực thi trong một giao dịch. Nếu có lỗi xảy ra, các thay đổi sẽ được rollback.
* ví dụ:

@Transactional

public void saveBook(Book book) {

bookRepository.save(book);

}

* loading ♾️

1. Định nghĩa các phương thức và các phương thức built-in trong JPA

* Các Built-in Method (tạo Query không cần viết thêm code)
* Spring JPA tự định nghĩa câu Query cho method bằng cách xử lý tên method. Cơ chế xây dựng truy vấn thông qua tên của method được quy định bởi các tiền tố find…By, read…By, query…By, count…By, và get…By, findDistinct, IgnoreCase , OrderBy
* save(S entity): Lưu (insert hoặc update) một thực thể vào cơ sở dữ liệu.

Ví dụ : save(book) truyền vào, saveBook(book)

* saveAll(Iterable<S> entities): Lưu nhiều thực thể vào cơ sở dữ liệu.
* findById(ID id): Tìm một thực thể theo ID.

Ví Dụ 👍

// version rút gọn

Person findByLastname(String lastname);

// verson đầy đủ

Person findPersonByLastname(String lastname);

Person findAllByLastname(String lastname);

* findAll(): Lấy tất cả các thực thể từ bảng.
* findAllById(Iterable<ID> ids): Lấy tất cả các thực thể theo danh sách ID.
* deleteById(ID id): Xóa một thực thể theo ID.
* delete(T entity): Xóa một thực thể cụ thể.
* deleteAll(Iterable<? extends T> entities): Xóa tất cả các thực thể trong danh sách.
* count(): Trả về số lượng thực thể trong bảng.
* existsById(ID id): Kiểm tra xem một thực thể có tồn tại dựa trên ID hay không.
* findDistinct…By…Or…

Ví dụ: findDistinctPeopleByLastnameOrFirstname

* find…DistinctBy…Or…

Ví du: findPeopleDistinctByLastnameOrFirstname

* findBy…And…AllIgnoreCase(String…, String…)

Ví dụ: findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname)

* findBy…OrderBy…Asc/Desc
* loading ♾️
* @Query
* Sử dụng câu truy vấn JPQL (Hibernate) và raw SQL
* Truyền tham số là gọi theo thứ tự các tham số của method bên dưới ?1, ?2

// Khi được gắn @Query, thì tên của method không còn tác dụng nữa

// Đây là JPQL

@Query("select u from User u where u.emailAddress = ?1")

User myCustomQuery(String emailAddress);

// Đây là Native SQL

@Query(value = "select \* from User u where u.email\_address = ?1", nativeQuery = true)

User myCustomQuery2(String emailAddress);

* Đặt tên cho tham số

// JPQL

@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")

User findUserByNamedParams(@Param("status") Integer status, @Param("name") String name);

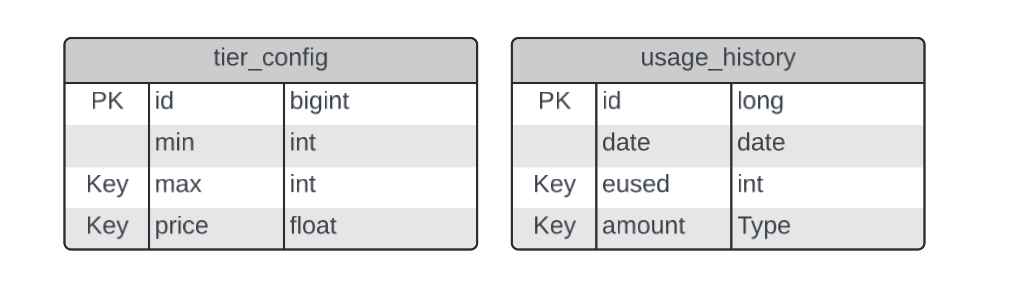
// Native SQL

@Query(value = "SELECT \* FROM Users u WHERE u.status = :status and u.name = :name", nativeQuery = true)

User findUserByNamedParamsNative(@Param("status") Integer status, @Param("name") String name);

1. loading ♾️
2. Thực hành:
3. Sử dụng postman ===> đã biết dùng trước đó :v hehe
4. Xây dựng một RESTful API đơn giản cho một ứng dụng quản lý tiền điện kia theo giá bậc thang
5. Ý tưởng giải quyết bài toán

* ý tưởng về DB : tạo 1 bảng quản lý bậc tiền điện, 1 bảng usage history để lưu các số điện đã sử dụng (như ảnh)



* xử lý logic tính tiền điện (main)
* tạo bậc tiền điện -> đưa thành list
* cho số điện sử dụng lặp qua list bậc điện

===> 1 biến lưu số điện còn lại (khởi giá trị = số điện)

===> 1 biến lưu số điện trong bậc đang duyệt có giá trị là min của số điện còn lại so với số điện của bậc đang duyệt

===> giá tiền điện thì bằng số điện trong bậc đó \* giá bậc đó

===> trừ số điện còn lại với số điện trong bậc đến khi = 0 thì break

* loading ♾️

1. Triển khai code

* tạo entity (tạo thuộc tính, constructer, getter, setter) trong project trên git
* repository
* tạo repo tier config với hàm get all được sort

public interface TierConfigRepository extends JpaRepository<TierConfig, Long>{

// xếp tăng dần để tí dùng trong usageHistoryService còn duyệt qua để tính xiền

List<TierConfig> findAllByOrderByMinUsageAsc();

}

* loading ♾️
* service với controller thì cũng chỉ là get, thêm, xoá như mấy task kia thôi º<º
* loading ♾️

1. loading ♾️
2. Mấy cái mới tìm hiểu được trong quá trình tìm hiểu task này
3. 🙂 bữa tánh test JPA nên tạo cái DB car để test tạm nên e name car, sau tạo nhiều entity vào 1 project luôn nên nó thập cẩm ===> phát sinh việc đổi tên DB

* vì e cài mysql qua docker nên cách đổi qua docker trước
* Truy cập vào container MySQL

docker exec -it <container\_name\_or\_id> bash

* Sử dụng mysqldump để sao lưu cơ sở dữ liệu:

mysqldump -u root -p car > /tmp/car\_backup.sql

* Đăng nhập vào MySQL:

mysql -u root -p

* Tạo cơ sở dữ liệu mới

CREATE DATABASE learn\_spring\_boot;

* Khôi phục dữ liệu từ tệp sao lưu

mysql -u root -p learn\_spring\_boot < /tmp/car\_backup.sql

* với mysql cài trên máy
* Tạo bản sao lưu database cũ

mysqldump -u [username] -p[password] old\_database\_name > old\_database\_name.sql

* Tạo database mới với tên mới

CREATE DATABASE new\_database\_name;

* Khôi phục dữ liệu vào database mới

mysql -u [username] -p[password] new\_database\_name < old\_database\_name.sql

* loading ♾️

1. loading ♾️
2. loading ♾️