**Training Backend-Web**

**2. Các câu lệnh cơ bản sql**

* Thao tác dữ liệu:

+, Select: Liệt kê các thuộc tính mong muốn trong truy vấn

+, Insert into: Chèn bản ghi mới vào bảng

+, Insert into select: Sao chép dữ liệu từ 1 bảng và thêm vào 1 bảng khác

+, Update: Sửa đổi các bản ghi hiện tại

+, Delete: Xoá các bản ghi hiện tại

* Định nghĩa dữ liệu:

+, Create table: tao bang, Drop table: xoa bang, Alter table: sửa các cột trong bảng

+, Create view: Tạo ra 1 truy vấn ảo không lưu dữ liệu mà lưu định nghĩa truy vấn, khi select từ view sẽ thực thi truy vấn gốc

+, Drop view: Xoa view do

* Trigger, Transaction:

+, Trigger(Insert, Update, Delete): là đoạn mã sql được tự động thực thi khi có 1 sự kiện cụ thể xảy ra trên 1 bảng hoặc 1 view

+, Transaction: Cho phép thực hiện nhiều lệnh và rollback lại nếu quá trình thực hiện xảy ra lỗi

+, Create procedure: tạo 1 thủ tục, từ đó có thể gọi khối sql nhiều lần

+, Create function: tạo ra 1 hàm người dùng tự định nghĩa, trả về giá trị sau khi thực hiện tính toán

+, Commit: Xác nhận toàn bộ các thao tác trong transaction sẽ được lưu vĩnh viễn vào csdl

+, Rollback: huỷ bỏ toàn bộ các thao tác, quay trở lại từ lúc bắt đầu transaction

* Điều khiển truy cập:

+, Grant: Cấp quyền cho người dùng

+, Revoke: Thu hồi quyền

**3. OOP trong Java**

* Class : là khuôn mẫu mô tả thuộc tính và hành vi của đối tượng
* Object: là thực thể được tạo ra từ class
* Biến: local variable, instance variable, static variable
* Hàm: gồm 3 thành phần: access modifier, return type, method name, và có thể có biến truyền vào
* Nhập xuất cơ bản: sử dụng scanner
* Nhập xuất file:

+, Với file input text thông thường: sử dụng FIleReader và BufferReader

+, Với file input nhị phân: sử dụng FIleInputStream và ObjectInputStream

* Các tính chất của OOP:

+, Đóng gói:

* Là việc che dấu dữ liệu bên trong class, chỉ có thể lấy và sửa thông qua getter, setter giúp kiểm soát dữ liệu khi nhập vào tránh bị thay đổi trực tiếp từ bên ngoài

+, Kế thừa:

* Là các class con kế thừa các tính chất, thuộc tính của class cha từ đó giúp tái sử dụng mã và linh hoạt trong việc mở rộng

+, Đa hình:

* Overloading( nạp chồng): Cùng 1 tên hàm nhưng khác tham số
* Overriding( ghi đè): Class con ghi đè các thuộc tính của class cha

+, Trừu tượng: là việc ẩn đi các chi tiết cài đặt, chỉ hiển thị các hành vi cần thiết

* Abstract class:

+, Sử dụng extend để kế thừa

+, 1 lớp con chỉ có thể kế thừa 1 lớp cha

+, Có abstract method và non-abstract method

+, Có thể có biến instance

+, Có constructor

+, Các class có quan hệ is-a chia sẻ code chung

* Interface:

+, Sử dụng implements để kế thừa

+, Có thể implements nhiều interface

+, Mặc định là abstract có thêm default hoặc static

+, Chỉ chứa được hàng số

+, Không có constructor

+, Định nghĩa hành vi mà nhiều class khác nhau có thể chia sẻ

**4. Dependency Injection(DI) và Inversion of Control(Ioc)**

**Dependency Injection(DI)**

* Trong oop các class được liên kết với nhau theo 1 quan hệ nào đó. Dependency là 1 loại quan hệ giữa 2 class mà trong đó 1 class hoạt động độc lập, 1 class hoạt động phụ thuộc class kia ⇒ Sự phụ thuộc chặt chẽ này gây nhiều khó khăn khi cần thay đổi, nâng cấp

⇒ Sử dụng design pattern Dependency Injection để giải quyết sự phụ thuộc này

* Theo nguyên tắc D( DIP - Dependency Inversion Principle) nguyên lý đảo ngược sự phụ thuộc:

+, Các module cấp cao không nên phụ thuộc vào chi tiết các module thấp. Cả 2 nên phụ thuộc vào abstract

+, Các class giao tiếp với nhau thông qua interface mà không phải thông qua implementions

* Dependency Injection (DI) là 1 design pattern, một kỹ thuật cho phép xoá bỏ sự phụ thuộc giữa các module, làm cho ứng dụng dễ dàng hơn trong việc bảo trì và thay đổi module
* Nhiệm vụ của dependency injection:

+, Tạo các đối tượng

+, Quản lý sự phụ thuộc(dependencies) giữa các đối tượng

+, Cung cấp(inject) các phụ thuộc được yêu cầu cho đối tượng( được truyền từ bên ngoài đối tượng)

* Nguyên tắc hoạt động:

+, Các module không giao tiếp trực tiếp với nhau mà thông qua interface. Module cấp thấp sẽ implements interface, module cấp cao se gọi module cấp thấp thông qua interface

+, Việc Module nào gắn với interface nào sẽ được config trong file properties, XML hoặc thông qua Annotation. trong Springboot là @Autowired, trong CDI là @Inject

vd:

Ở Service có interface buildingService, 1 class buildingServiceImpl được implements interface đó

⇒ Khi đó trong Controller chỉ cần autowired buildingService mà không cần buildingServiceImpl bởi nó đã được inject vào interface đó

vd: Trong java spring boot

* interface

public interface ICategoryService {

Category createCategory(CategoryDTO category);

Category getCategoryById(long id);

List<Category> getAllCategories();

Category updateCategory(long categoryId, CategoryDTO category);

void deleteCategory(long id);

}

* Class implement interface đó

@Service

public class CategoryService implements ICategoryService {

@Autowired

private CategoryRepository categoryRepository;

@Override

public Category createCategory(CategoryDTO categoryDTO) {

Category newCategory = Category

.*builder*()

.name(categoryDTO.getName())

.build();

return categoryRepository.save(newCategory);

}

@Override

public Category getCategoryById(long id) {

return categoryRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Category not found"));

}

@Override

public List<Category> getAllCategories() {

return categoryRepository.findAll();

}

@Override

public Category updateCategory(long categoryId,

CategoryDTO categoryDTO) {

Category existingCategory = getCategoryById(categoryId);

existingCategory.setName(categoryDTO.getName());

categoryRepository.save(existingCategory);

return existingCategory;

}

@Override

public void deleteCategory(long id) {

//xóa xong

categoryRepository.deleteById(id);

}

}

* Controller sẽ autowired interface đó ra:

@RestController

@RequestMapping("${api.prefix}/categories")

public class CategoryController {

@Autowired

private ICategoryService categoryService;

@GetMapping("")

public ResponseEntity<List<Category>> getAllCategories(

@RequestParam("page") int page,

@RequestParam("limit") int limit

) {

List<Category> categories = categoryService.getAllCategories();

return ResponseEntity.*ok*(categories);

}

@PostMapping("")

//Nếu tham số truyền vào là 1 object thì sao ? => Data Transfer Object = Request Object

public ResponseEntity<?> createCategory(

@Valid @RequestBody CategoryDTO categoryDTO,

BindingResult result) {

if(result.hasErrors()) {

List<String> errorMessages = result.getFieldErrors()

.stream()

.map(FieldError::getDefaultMessage)

.toList();

return ResponseEntity.*badRequest*().body(errorMessages);

}

categoryService.createCategory(categoryDTO);

return ResponseEntity.*ok*("Insert category successfully");

}

* Ứng dụng Dependency Injection:

+, Khi cần inject các giá trị từ 1 cấu hình cho 1 hoặc nhiều module khác

+, Khi cần inject 1 dependency cho nhiều module khác nhau

+, Khi cần tách biệt các dependency giữa các môi trường phát triển khác nhau. vd với môi trường dev thì chỉ cần log gửi mail, trong môi trường product cần gửi mail thông qua api thực sự

vd:

public interface EmailService {

void sendEmail(String to, String subject, String body);

}

import org.springframework.context.annotation.Profile;

import org.springframework.stereotype.Service;

@Service

@Profile("dev")

public class DevEmailService implements EmailService {

@Override

public void sendEmail(String to, String subject, String body) {

System.out.println("[DEV] Fake send email to: " + to);

}

}

import org.springframework.context.annotation.Profile;

import org.springframework.stereotype.Service;

@Service

@Profile("prod")

public class ProdEmailService implements EmailService {

@Override

public void sendEmail(String to, String subject, String body) {

System.out.println("[PROD] Real email sent to: " + to);

}

}

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

import org.springframework.stereotype.Service;

@Service

public class UserService {

@Autowired

private EmailService emailService;

public void register(String email) {

System.out.println("Registering " + email);

emailService.sendEmail(email, "Welcome", "Thanks for registering!");

}

}

**Inversion of Control(Ioc)**

* Inversion of control : Đảo ngược quyền điều khiển, là 1 trong nguyên lý cốt lõi của oop và spring framwork. Thay vì chính chương trình điều khiển mọi thứ(tạo object, gọi method, quản lý vòng đời) thì 1 container/framework khác sẽ làm điều đó

⇒ Bình thường tự new object rồi tự gọi hàm. Với Ioc chỉ cần định nghĩa quan hệ còn spring sẽ tự tạo object(bean), tự quản lý và bơm chúng vào nơi cần dùng

vd: java

class Engine {

void start() {

System.*out*.println("Engine started!");

}

}

class Car {

private Engine engine;

public Car() {

this.engine = new Engine(); // Car tự tạo Engine

}

public void drive() {

engine.start();

System.*out*.println("Car is moving...");

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car(); // tự new

car.drive();

}

}

⇒ Car phụ thuộc trực tiếp vào Engine, nếu cần thay đổi thì phải sửa code trong car nữa

Áp dụng Ioc + DI

interface Engine {

void start();

}

class GasEngine implements Engine {

public void start() {

System.*out*.println("Gas engine started!");

}

}

class ElectricEngine implements Engine {

public void start() {

System.*out*.println("Electric engine started!");

}

}

class Car {

private Engine engine;

// Dependency được inject từ bên ngoài (IoC)

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.*out*.println("Car is moving...");

}

}

public class Main {

public static void main(String[] args) {

// Container (ở đây chính bạn) chọn engine phù hợp

Engine engine = new ElectricEngine(); // hoặc GasEngine

Car car = new Car(engine); // Inject dependency

car.drive();

}

}

⇒ Giờ car không phụ thuộc trực tiếp vào GasEngine nữa mà sẽ chỉ biết đến interface engine

⇒ Quyền điều khiến việc chọn loại Engine đã được đảo ra ngoài, đó chính là Ioc

* Trong spring framework:

import org.springframework.stereotype.Component;

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

import org.springframework.context.annotation.\*;

interface Engine {

void start();

}

@Component

class GasEngine implements Engine {

public void start() {

System.*out*.println("Gas engine started!");

}

}

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.*out*.println("Car is moving...");

}

}

@Configuration

@ComponentScan("com.example")

class AppConfig {}

public class Main {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Car car = context.getBean(Car.class);

car.drive();

}

}

⇒ Ioc là nguyên lý tổng quát: đảo ngược quyền điều khiển. DI là kỹ thuật cụ thể để thực hiện Ioc( tiêm dependency vào class). Nói các khác DI là cách thực hiện Ioc

* ApplicationContext là container chính trong Spring dùng để:

+, Tạo, quản lý và huỷ các Bean(đối tượng được spring quản lý) các class được khai báo với annotation @Component

+, Thực hiện Dependency Injection dựa trên cấu hình XML hoặc annotation

⇒ Là lõi của mọi ứng dụng Spring, Springboot

* Một số ApplicationContext:

+, ClassPathXmlApplicationContext: Đọc file cấu hình xml trong class path

+, FileSystemXmlApplicationContext: Đọc file Xml từ đường dẫn tuyệt đối

+, AnnotationConfigApplicationContext: Dùng cho Spring Java Config(@Configuration, @Bean)

+, WebApplicationContext: Dành cho ứng dung web spring MVC