# Spring boost

I**.Spring Boot là gì?**

* 1. Là một framework mở rộng của Spring, giúp tạo ra các ứng dụng Java một cách nhanh chóng và dễ dàng mà không cần cấu hình phức tạp.

**Auto-Configuration:**

* 1. Spring Boot tự động cấu hình ứng dụng dựa trên các thư viện có trong classpath.

**Starter Dependencies:**

* 1. Các thư viện đã được định nghĩa sẵn để dễ dàng tích hợp (ví dụ: spring-boot-starter-web, spring-boot-starter-data-jpa).

**Application Properties:**

* 1. Tập tin cấu hình (application.properties hoặc application.yml) để cấu hình ứng dụng, như cổng, thông tin kết nối cơ sở dữ liệu.

**Spring Boot CLI:**

* 1. Cung cấp một công cụ dòng lệnh để chạy và kiểm tra các ứng dụng Spring Boot.

**RESTful APIs:**

* 1. Hỗ trợ phát triển các dịch vụ RESTful thông qua các annotation như @RestController, @RequestMapping, @GetMapping, @PostMapping.

**Microservices:**

* 1. Spring Boot hỗ trợ phát triển các dịch vụ nhỏ, độc lập, dễ dàng triển khai và mở rộng.

**Spring Data JPA:**

* 1. Cung cấp khả năng làm việc với cơ sở dữ liệu thông qua các repository mà không cần viết nhiều mã SQL.

**Security:**

* 1. Spring Security cho phép thêm tính năng bảo mật cho ứng dụng (xác thực, phân quyền).

**Actuator:**

* 1. Cung cấp các endpoint để quản lý và giám sát ứng dụng, như kiểm tra trạng thái, thông tin về các bean, v.v.

**Testing:**

* 1. Hỗ trợ viết các bài kiểm tra cho ứng dụng với các annotation như @SpringBootTest, @MockBean.

**Containerization:**

* 1. Hỗ trợ tích hợp với Docker để triển khai ứng dụng một cách dễ dàng.

**External Configuration:**

* 1. Hỗ trợ cấu hình ứng dụng từ các nguồn bên ngoài (như file, database, hoặc môi trường) thông qua Spring Cloud Config.

II.Cấu trúc một dự án Spring Boot thông thường được tổ chức theo cách sau, nhằm giúp dễ quản lý và mở rộng trong quá trình phát triển phần mềm. Dưới đây là một cấu trúc cơ bản mà bạn có thể gặp trong dự án Spring Boot:

### 1. ****Root Folder (Thư mục gốc)****:

* pom.xml **hoặc** build.gradle: Tập tin này dùng để quản lý các dependency và build dự án. Nếu dùng Maven thì bạn sẽ có pom.xml, còn dùng Gradle thì sẽ có build.gradle.
* src/main/java: Chứa mã nguồn chính của ứng dụng.
* src/main/resources: Chứa các tệp cấu hình và tài nguyên tĩnh như tệp cấu hình application.properties hoặc application.yml, các template, và file static như CSS, JS.

### 2. ****Thư mục Java (****src/main/java****)****:

Đây là thư mục chứa toàn bộ mã nguồn của ứng dụng. Các thành phần trong ứng dụng thường được chia ra theo từng lớp logic khác nhau:

* com.example.projectname: Đây là package chính của ứng dụng. Thông thường sẽ theo dạng com.companyname.projectname.
  + controller: Chứa các lớp điều khiển (Controller) xử lý các request HTTP và trả về response. Ví dụ, RESTful controllers hoặc web controllers.
  + service: Chứa các lớp service, nơi xử lý các logic nghiệp vụ của ứng dụng.
  + repository: Chứa các lớp repository, nơi tương tác với cơ sở dữ liệu (thường dùng với JPA hoặc CRUDRepository).
  + model **hoặc** entity: Chứa các lớp mô hình (Model) đại diện cho dữ liệu trong cơ sở dữ liệu, các entity.
  + dto (Data Transfer Object - tùy chọn): Chứa các lớp DTO, để truyền dữ liệu giữa các tầng của ứng dụng.
  + config (tùy chọn): Chứa các lớp cấu hình (configuration), như cấu hình bảo mật, cơ sở dữ liệu, hoặc cấu hình chung cho các thành phần khác.

Ví dụ cụ thể cấu trúc thư mục Java:

css

Copy code

src/main/java

└── com

└── example

└── projectname

├── controller

├── service

├── repository

├── model

└── config

### 3. ****Thư mục Resources (****src/main/resources****)****:

* application.properties **hoặc** application.yml: Tệp cấu hình của ứng dụng, nơi bạn cấu hình các thông số như cổng chạy ứng dụng, thông tin kết nối cơ sở dữ liệu, hoặc cấu hình bảo mật.
* static/: Chứa các tài nguyên tĩnh như CSS, JS, hoặc hình ảnh. Các file ở đây được truy cập trực tiếp thông qua URL.
* templates/: Thường dùng trong các ứng dụng Spring MVC, chứa các template HTML (ví dụ: Thymeleaf, Freemarker) để render giao diện.
* messages.properties: Tệp chứa các chuỗi localization, dùng cho đa ngôn ngữ.

Ví dụ cấu trúc thư mục Resources:

css

src/main/resources

├── application.properties

├── static

│ └── css

├── templates

└── messages.properties

### 4. ****Thư mục Test (****src/test/java****)****:

* Chứa các lớp kiểm thử (test classes) để viết các bài kiểm thử đơn vị (unit tests) hoặc kiểm thử tích hợp (integration tests).

### 5. ****Thành phần chính trong cấu trúc dự án****:

* **Main Application Class**: Đây là lớp chính, thường nằm ở package gốc của ứng dụng, được đánh dấu bằng annotation @SpringBootApplication. Đây là nơi bắt đầu quá trình chạy ứng dụng Spring Boot.

Ví dụ:

@SpringBootApplicationpublic class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

### 6. ****Các phụ thuộc phổ biến****:

Trong tập tin pom.xml (với Maven) hoặc build.gradle (với Gradle), bạn sẽ khai báo các dependency cần thiết như:

* **Spring Web**: để tạo RESTful web services.
* **Spring Data JPA**: để làm việc với cơ sở dữ liệu.
* **Spring Security**: để bảo mật ứng dụng.
* **Thymeleaf**: để tạo giao diện web động.

III.com.example.projectname: Đây là package chính của ứng dụng. Thông thường sẽ theo dạng com.companyname.projectname. controller: Chứa các lớp điều khiển (Controller) xử lý các request HTTP và trả về response. Ví dụ, RESTful controllers hoặc web controllers. service: Chứa các lớp service, nơi xử lý các logic nghiệp vụ của ứng dụng. repository: Chứa các lớp repository, nơi tương tác với cơ sở dữ liệu (thường dùng với JPA hoặc CRUDRepository). model hoặc entity: Chứa các lớp mô hình (Model) đại diện cho dữ liệu trong cơ sở dữ liệu, các entity. dto (Data Transfer Object - tùy chọn): Chứa các lớp DTO, để truyền dữ liệu giữa các tầng của ứng dụng. config (tùy chọn): Chứa các lớp cấu hình (configuration), như cấu hình bảo mật, cơ sở dữ liệu, hoặc cấu hình chung cho các thành phần khác. đưa ra ví dụ code

Dưới đây là một ví dụ cơ bản về cách tổ chức các lớp trong một dự án Spring Boot theo cấu trúc mà bạn đã đề cập, bao gồm **controller**, **service**, **repository**, **model/entity**, **dto**, và **config**:

### 1. model ****(hoặc**** entity****)**** - Định nghĩa lớp mô hình

package com.example.projectname.model;

import javax.persistence.Entity;import javax.persistence.GeneratedValue;import javax.persistence.GenerationType;import javax.persistence.Id;

@Entitypublic class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

### 2. repository - Tương tác với cơ sở dữ liệu

package com.example.projectname.repository;

import com.example.projectname.model.User;import org.springframework.data.jpa.repository.JpaRepository;import org.springframework.stereotype.Repository;

@Repositorypublic interface UserRepository extends JpaRepository<User, Long> {

// Bạn có thể thêm các phương thức tìm kiếm tùy chỉnh tại đây

}

### 3. service - Xử lý logic nghiệp vụ

package com.example.projectname.service;

import com.example.projectname.model.User;import com.example.projectname.repository.UserRepository;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;

import java.util.List;

@Servicepublic class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

public User saveUser(User user) {

return userRepository.save(user);

}

public void deleteUser(Long id) {

userRepository.deleteById(id);

}

}

### 4. controller - Xử lý các yêu cầu HTTP

package com.example.projectname.controller;

import com.example.projectname.dto.UserDto;import com.example.projectname.model.User;import com.example.projectname.service.UserService;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController@RequestMapping("/api/users")public class UserController {

@Autowired

private UserService userService;

@GetMapping

public List<User> getAllUsers() {

return userService.getAllUsers();

}

@GetMapping("/{id}")

public User getUserById(@PathVariable Long id) {

return userService.getUserById(id);

}

@PostMapping

public User createUser(@RequestBody UserDto userDto) {

User user = new User();

user.setName(userDto.getName());

user.setEmail(userDto.getEmail());

return userService.saveUser(user);

}

@DeleteMapping("/{id}")

public void deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

}

}

### 5. dto (tùy chọn) - Đối tượng truyền dữ liệu

package com.example.projectname.dto;

public class UserDto {

private String name;

private String email;

// Getters and Setters

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

### 6. config (tùy chọn) - Cấu hình cho ứng dụng

package com.example.projectname.config;

import org.springframework.context.annotation.Bean;import org.springframework.context.annotation.Configuration;import org.springframework.web.servlet.config.annotation.CorsRegistry;import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configurationpublic class WebConfig {

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurer() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*")

.allowedOrigins("http://localhost:3000");

}

};

}

}

### 7. Main Application Class

package com.example.projectname;

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplicationpublic class ProjectnameApplication {

public static void main(String[] args) {

SpringApplication.run(ProjectnameApplication.class, args);

}

}

### Tóm tắt:

* User là entity đại diện cho dữ liệu người dùng.
* UserRepository là interface để làm việc với cơ sở dữ liệu.
* UserService xử lý logic nghiệp vụ.
* UserController nhận và trả về phản hồi cho các request HTTP.
* UserDto là lớp DTO dùng để truyền dữ liệu giữa các tầng.
* WebConfig là lớp cấu hình cho CORS (để cho phép kết nối từ các miền khác).

Cấu trúc này giúp chia rõ ràng các thành phần khác nhau, làm cho mã nguồn dễ duy trì và mở rộng.

IV.post man

Postman là một công cụ mạnh mẽ dùng để kiểm thử và phát triển API. Nó cho phép bạn gửi các yêu cầu HTTP, kiểm tra phản hồi từ server và thao tác với các API mà không cần phải viết mã phức tạp. Dưới đây là cách sử dụng Postman để kiểm thử một số API cơ bản từ ứng dụng Spring Boot mà mình đã giới thiệu ở trên.

### 1. ****Cài đặt Postman****

Có thể tải Postman từ trang web chính thức: Postman Download.

### 2. ****Gửi Request tới API****

Giả sử đã chạy ứng dụng Spring Boot và API đang hoạt động trên http://localhost:8080. Sau đó, bạn có thể kiểm tra các chức năng như thêm người dùng (POST), lấy danh sách người dùng (GET), cập nhật (PUT), hoặc xóa (DELETE) người dùng thông qua Postman.

#### a. ****Tạo một User mới (POST Request)****

1. Mở Postman, chọn POST từ menu thả xuống ở góc trên bên trái.
2. Nhập URL của API: http://localhost:8080/api/users.
3. Chuyển sang tab **Body**, chọn **raw** và sau đó chọn **JSON** từ dropdown.
4. Nhập dữ liệu của user vào dưới dạng JSON, ví dụ:

{

"name": "John Doe",

"email": "johndoe@example.com"}

1. Nhấn nút **Send**. Nếu yêu cầu thành công, bạn sẽ nhận được phản hồi có chứa thông tin user vừa tạo và HTTP Status 200 OK.

#### b. ****Lấy danh sách các User (GET Request)****

1. Chọn GET từ menu thả xuống.
2. Nhập URL: http://localhost:8080/api/users.
3. Nhấn **Send**.
4. Bạn sẽ nhận được phản hồi dưới dạng JSON chứa danh sách các user, ví dụ:

[

{

"id": 1,

"name": "John Doe",

"email": "johndoe@example.com"

},

{

"id": 2,

"name": "Jane Smith",

"email": "janesmith@example.com"

}]

#### c. ****Lấy thông tin User theo ID (GET Request)****

1. Chọn GET từ menu thả xuống.
2. Nhập URL: http://localhost:8080/api/users/1 (trong đó 1 là ID của user).
3. Nhấn **Send**. Bạn sẽ nhận được thông tin chi tiết về user có ID là 1.

{

"id": 1,

"name": "John Doe",

"email": "johndoe@example.com"}

#### d. ****Xóa một User (DELETE Request)****

1. Chọn DELETE từ menu thả xuống.
2. Nhập URL: http://localhost:8080/api/users/1.
3. Nhấn **Send**. Nếu user có ID là 1 tồn tại, user đó sẽ bị xóa và bạn sẽ nhận được phản hồi HTTP Status 200 OK.

### 3. ****Lưu và Quản lý các Request****

* Bạn có thể lưu các request thành **Collections** trong Postman để tái sử dụng sau này. Các Collection này sẽ giúp bạn tổ chức các yêu cầu API theo nhóm và dễ dàng truy cập chúng trong các lần kiểm thử sau.

### 4. ****Cấu hình môi trường****

Postman cũng hỗ trợ cấu hình môi trường (environments), nơi bạn có thể định nghĩa các biến như URL cơ sở của API (http://localhost:8080). Điều này giúp bạn dễ dàng thay đổi khi chuyển từ môi trường phát triển sang môi trường sản xuất.

Ví dụ, bạn có thể cấu hình:

* **Dev environment** với URL là http://localhost:8080.
* **Prod environment** với URL là https://api.example.com.

Khi sử dụng, bạn chỉ cần chuyển đổi giữa các môi trường mà không cần thay đổi URL trong mỗi request.

### 5. ****Kiểm tra tự động hóa (Automation Test)****

Postman còn hỗ trợ bạn viết script để kiểm thử tự động bằng cách thêm các đoạn mã kiểm tra trong tab **Tests** của mỗi request. Ví dụ:

pm.test("Status code is 200", function () {

pm.response.to.have.status(200);

});

Sau khi gửi yêu cầu, Postman sẽ kiểm tra liệu mã trạng thái HTTP có phải là 200 hay không và thông báo kết quả.

Như vậy, với Postman, bạn có thể dễ dàng kiểm tra và làm việc với các API mà không cần phải triển khai giao diện người dùng, giúp tiết kiệm thời gian và tăng hiệu quả phát triển.

Dưới đây là một ví dụ về một **Spring Boot Controller** đơn giản để quản lý thông tin người dùng. Trong ví dụ này, controller chỉ có các phương thức cơ bản để thực hiện các thao tác CRUD (Create, Read, Update, Delete) thông qua các request HTTP.

### Ví dụ đơn giản về một UserController:

#### 1. ****Entity User****:

package com.example.projectname.model;

import javax.persistence.Entity;import javax.persistence.GeneratedValue;import javax.persistence.GenerationType;import javax.persistence.Id;

@Entitypublic class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

#### 2. ****Repository UserRepository****:

package com.example.projectname.repository;

import com.example.projectname.model.User;import org.springframework.data.jpa.repository.JpaRepository;import org.springframework.stereotype.Repository;

@Repositorypublic interface UserRepository extends JpaRepository<User, Long> {

}

#### 3. ****Service UserService****:

package com.example.projectname.service;

import com.example.projectname.model.User;import com.example.projectname.repository.UserRepository;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;

import java.util.List;

@Servicepublic class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

public User saveUser(User user) {

return userRepository.save(user);

}

public void deleteUser(Long id) {

userRepository.deleteById(id);

}

}

#### 4. ****Controller UserController****:

package com.example.projectname.controller;

import com.example.projectname.model.User;import com.example.projectname.service.UserService;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController@RequestMapping("/api/users")public class UserController {

@Autowired

private UserService userService;

// Lấy tất cả user

@GetMapping

public List<User> getAllUsers() {

return userService.getAllUsers();

}

// Lấy user theo ID

@GetMapping("/{id}")

public User getUserById(@PathVariable Long id) {

return userService.getUserById(id);

}

// Tạo mới user

@PostMapping

public User createUser(@RequestBody User user) {

return userService.saveUser(user);

}

// Xóa user theo ID

@DeleteMapping("/{id}")

public void deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

}

}

### Giải thích:

@RestController: Annotation này cho biết đây là một REST Controller. Tất cả các phương thức trong controller sẽ trả về dữ liệu JSON.

@RequestMapping("/api/users"): Tất cả các endpoint trong controller này sẽ bắt đầu với đường dẫn /api/users.

**Các phương thức chính**:

* + @GetMapping: Phương thức getAllUsers() sẽ trả về danh sách tất cả các người dùng dưới dạng JSON.
  + @GetMapping("/{id}"): Phương thức getUserById(Long id) sẽ trả về thông tin của người dùng theo ID.
  + @PostMapping: Phương thức createUser() sẽ nhận dữ liệu từ body của request dưới dạng JSON để tạo mới người dùng.
  + @DeleteMapping("/{id}"): Phương thức deleteUser() sẽ xóa người dùng theo ID.

### Ví dụ sử dụng với Postman:

**GET** /api/users: Lấy danh sách tất cả người dùng.

**GET** /api/users/{id}: Lấy thông tin người dùng có ID là {id}.

**POST** /api/users: Tạo mới người dùng. Dữ liệu truyền vào body dưới dạng JSON:

{

"name": "John Doe",

"email": "johndoe@example.com"}

**DELETE** /api/users/{id}: Xóa người dùng có ID là {id}.Nắm vững các design pattern cơ bản của Java

V.Các **design pattern** trong Java là các giải pháp thiết kế đã được thử nghiệm và kiểm chứng, giúp giải quyết các vấn đề thường gặp trong lập trình hướng đối tượng. Nắm vững các pattern này sẽ giúp bạn viết mã dễ bảo trì, linh hoạt và hiệu quả hơn. Dưới đây là những pattern cơ bản mà bạn nên nắm vững:

### 1. ****Creational Patterns (Nhóm tạo đối tượng)****

**Singleton**: Đảm bảo rằng chỉ có một thể hiện (instance) của một lớp được tạo ra và cung cấp một điểm truy cập toàn cục cho nó.

* + **Ví dụ**: Lớp quản lý kết nối cơ sở dữ liệu.

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

**Factory Method**: Cung cấp một giao diện để tạo đối tượng, nhưng để các lớp con quyết định lớp nào sẽ được khởi tạo.

* + **Ví dụ**: Tạo các đối tượng hình học như Circle, Square.

public abstract class ShapeFactory {

public abstract Shape createShape();

}

public class CircleFactory extends ShapeFactory {

public Shape createShape() {

return new Circle();

}

}

**Abstract Factory**: Tạo ra một hệ thống các Factory liên quan đến nhau mà không cần chỉ rõ lớp cụ thể nào sẽ được khởi tạo.

* + **Ví dụ**: Tạo các họa tiết GUI cho hệ điều hành khác nhau như Windows và macOS.

**Builder**: Tách quá trình xây dựng một đối tượng phức tạp ra khỏi biểu diễn của nó, cho phép tạo các đối tượng khác nhau bằng cùng một tiến trình xây dựng.

* + **Ví dụ**: Tạo đối tượng House với nhiều thuộc tính khác nhau.

public class House {

private String foundation;

private String structure;

private String roof;

private House(Builder builder) {

this.foundation = builder.foundation;

this.structure = builder.structure;

this.roof = builder.roof;

}

public static class Builder {

private String foundation;

private String structure;

private String roof;

public Builder foundation(String foundation) {

this.foundation = foundation;

return this;

}

public Builder structure(String structure) {

this.structure = structure;

return this;

}

public Builder roof(String roof) {

this.roof = roof;

return this;

}

public House build() {

return new House(this);

}

}

}

**Prototype**: Dùng để khởi tạo đối tượng bằng cách sao chép (clone) từ một đối tượng đã có thay vì tạo mới.

* + **Ví dụ**: Tạo các bản sao của đối tượng người dùng hoặc cấu trúc dữ liệu.

public class Prototype implements Cloneable {

@Override

public Prototype clone() throws CloneNotSupportedException {

return (Prototype) super.clone();

}

}

### 2. ****Structural Patterns (Nhóm cấu trúc)****

**Adapter**: Cho phép các lớp có giao diện không tương thích làm việc cùng nhau bằng cách sử dụng một lớp chuyển đổi (adapter).

* + **Ví dụ**: Chuyển đổi đối tượng USB thành Type-C để kết nối.

public interface USB {

void connectWithUsbCable();

}

public class UsbAdapter implements TypeC {

private USB usbDevice;

public UsbAdapter(USB usbDevice) {

this.usbDevice = usbDevice;

}

@Override

public void connectWithTypeCCable() {

usbDevice.connectWithUsbCable();

}

}

**Bridge**: Tách rời phần trừu tượng (abstraction) ra khỏi phần thực thi (implementation) để cả hai có thể thay đổi độc lập nhau.

* + **Ví dụ**: Giao diện người dùng có thể thay đổi màu sắc mà không ảnh hưởng đến chức năng hiển thị.

**Composite**: Cho phép bạn nhóm các đối tượng lại thành cấu trúc cây để làm việc với chúng như với các đối tượng đơn lẻ.

* + **Ví dụ**: Hệ thống quản lý tập tin, nơi cả thư mục và tập tin đều có thể được xử lý đồng nhất.

**Decorator**: Cho phép thêm hành vi hoặc trạng thái mới cho các đối tượng trong khi vẫn giữ nguyên giao diện ban đầu.

* + **Ví dụ**: Thêm tính năng "đường viền" hoặc "đổ bóng" cho các thành phần giao diện người dùng.

public class Coffee {

public double cost() {

return 5;

}

}

public class MilkDecorator extends Coffee {

private Coffee coffee;

public MilkDecorator(Coffee coffee) {

this.coffee = coffee;

}

@Override

public double cost() {

return coffee.cost() + 2;

}

}

**Facade**: Cung cấp một giao diện đơn giản để tương tác với một hệ thống phức tạp gồm nhiều lớp và thành phần khác nhau.

* + **Ví dụ**: Một lớp giao diện đơn giản cho hệ thống quản lý ngân hàng, như xử lý khoản vay, tài khoản, và thanh toán.

**Proxy**: Cung cấp một đối tượng đại diện (proxy) kiểm soát việc truy cập vào đối tượng thật.

* + **Ví dụ**: Proxy cho kết nối internet để kiểm soát quyền truy cập và tăng hiệu suất.

### 3. ****Behavioral Patterns (Nhóm hành vi)****

**Observer**: Cho phép một đối tượng (subject) thông báo cho các đối tượng quan sát (observers) khi có sự thay đổi trạng thái.

* + **Ví dụ**: Hệ thống theo dõi giá cổ phiếu, nơi nhiều người dùng có thể theo dõi giá cổ phiếu.

public interface Observer {

void update();

}

public class StockPriceObserver implements Observer {

@Override

public void update() {

System.out.println("Stock price updated!");

}

}

**Strategy**: Cho phép định nghĩa một họ thuật toán, đóng gói từng thuật toán, và làm cho chúng có thể thay đổi thay thế nhau.

* + **Ví dụ**: Lựa chọn phương thức thanh toán khác nhau (PayPal, thẻ tín dụng).

public interface PaymentStrategy {

void pay(int amount);

}

public class CreditCardStrategy implements PaymentStrategy {

@Override

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card");

}

}

**Command**: Đóng gói một yêu cầu thành đối tượng để có thể truyền tải yêu cầu này như một đối tượng thực thi sau.

* + **Ví dụ**: Hệ thống đặt hàng, nơi mỗi yêu cầu được xử lý dưới dạng một lệnh.

**State**: Cho phép một đối tượng thay đổi hành vi khi trạng thái nội tại của nó thay đổi.

* + **Ví dụ**: Máy bán hàng tự động thay đổi hành vi tùy vào trạng thái (đang chờ tiền, có hàng, hết hàng).

**Template Method**: Định nghĩa khung của một thuật toán trong một phương thức, nhưng cho phép các lớp con ghi đè lại một số bước mà không thay đổi cấu trúc chung.

* + **Ví dụ**: Quy trình nướng bánh, nơi từng bước có thể được điều chỉnh bởi người dùng.

### Tổng kết:

* **Creational Patterns** giúp tạo đối tượng theo cách linh hoạt và tái sử dụng.
* **Structural Patterns** giúp tổ chức các lớp và đối tượng một cách hợp lý.
* **Behavioral Patterns** giúp quản lý hành vi giữa các đối tượng trong một hệ thống.

Hiểu và áp dụng các design pattern này sẽ giúp bạn phát triển các hệ thống phần mềm có cấu trúc rõ ràng, dễ bảo trì và mở rộng.

VI.Cấu hình ứng dụng với application.properties

Trong **Spring Boot**, tệp cấu hình application.properties được sử dụng để cấu hình các tham số khác nhau của ứng dụng như cấu hình kết nối cơ sở dữ liệu, thiết lập cổng, định nghĩa các thông số môi trường, và nhiều tính năng khác.

Dưới đây là một số ví dụ về các thiết lập phổ biến trong tệp application.properties:

### 1. ****Cấu hình cổng cho ứng dụng****

* Thiết lập cổng mà ứng dụng Spring Boot sẽ chạy trên đó.

properties

server.port=8080

### 2. ****Cấu hình kết nối cơ sở dữ liệu (MySQL)****

* Thiết lập kết nối cơ sở dữ liệu, thông tin xác thực, và driver.

properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=password

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

### 3. ****Cấu hình Hibernate (JPA)****

* Thiết lập liên quan đến JPA và Hibernate.

properties

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

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

* spring.jpa.hibernate.ddl-auto=update: Tạo hoặc cập nhật các bảng trong cơ sở dữ liệu dựa trên các entity.
* spring.jpa.show-sql=true: Hiển thị câu lệnh SQL được thực thi trong console.

### 4. ****Cấu hình Logging****

* Thiết lập mức độ logging cho ứng dụng.

properties

logging.level.org.springframework=INFO

logging.level.com.example=DEBUG

* Bạn có thể điều chỉnh mức độ logging cho từng package hoặc lớp cụ thể.

### 5. ****Cấu hình profile cho môi trường****

* Sử dụng profiles để tách cấu hình cho các môi trường khác nhau (development, production, testing).

properties

spring.profiles.active=dev

Bạn có thể tạo các tệp application-dev.properties, application-prod.properties để lưu cấu hình tương ứng cho từng môi trường.

### 6. ****Cấu hình gửi email****

* Cấu hình thông tin gửi email qua SMTP.

properties

spring.mail.host=smtp.gmail.com

spring.mail.port=587

spring.mail.username=myemail@gmail.com

spring.mail.password=mypassword

spring.mail.properties.mail.smtp.auth=true

spring.mail.properties.mail.smtp.starttls.enable=true

### 7. ****Cấu hình thời gian timeout****

* Đặt thời gian timeout cho các request.

properties

server.connection-timeout=5000 # 5 giây

### 8. ****Cấu hình bảo mật****

* Tắt bảo mật CSRF trong trường hợp ứng dụng không cần bảo mật form.

properties

security.basic.enabled=false

spring.security.enable-csrf=false

### 9. ****Cấu hình thông số tuỳ chỉnh****

Bạn có thể thêm các thuộc tính tuỳ chỉnh cho ứng dụng và truy cập chúng thông qua Spring.

properties

app.name=My Spring Boot Application

app.version=1.0.0

Trong lớp Java, bạn có thể lấy các giá trị này bằng cách sử dụng @Value:

@Value("${app.name}")private String appName;

@Value("${app.version}")private String appVersion;

### 10. ****Cấu hình cache (Redis)****

properties

spring.cache.type=redis

spring.redis.host=localhost

spring.redis.port=6379

### 11. ****Cấu hình thư mục static****

Nếu bạn muốn thay đổi thư mục chứa các tài nguyên tĩnh (static), chẳng hạn như HTML, CSS, JS:

properties

spring.mvc.static-path-pattern=/resources/\*\*

spring.resources.static-locations=classpath:/static/,classpath:/public/

### Tóm tắt:

* **Tệp** application.properties là nơi bạn có thể dễ dàng cấu hình ứng dụng mà không cần phải sửa đổi mã nguồn.
* Spring Boot hỗ trợ nhiều cấu hình khác nhau từ cơ sở dữ liệu, bảo mật, logging, đến các cấu hình tùy chỉnh riêng cho từng dự án.

VII. Các câu lệnh trong sql

### 1. ****INSERT****

Câu lệnh **INSERT** được sử dụng để thêm dữ liệu mới vào bảng.

INSERT INTO table\_name (column1, column2, column3, ...)VALUES (value1, value2, value3, ...);

**Ví dụ**:

INSERT INTO employees (name, position, salary)VALUES ('John Doe', 'Developer', 5000);

### 2. ****UPDATE****

Câu lệnh **UPDATE** được sử dụng để sửa đổi dữ liệu hiện có trong bảng.

UPDATE table\_nameSET column1 = value1, column2 = value2, ...WHERE condition;

**Ví dụ**:

UPDATE employeesSET salary = 6000WHERE name = 'John Doe';

### 3. ****DELETE****

Câu lệnh **DELETE** được sử dụng để xóa dữ liệu trong bảng dựa trên điều kiện cụ thể.

DELETE FROM table\_nameWHERE condition;

**Ví dụ**:

DELETE FROM employeesWHERE name = 'John Doe';

### 4. ****SELECT****

Câu lệnh **SELECT** dùng để truy vấn và lấy dữ liệu từ bảng.

SELECT column1, column2, ...FROM table\_nameWHERE condition;

**Ví dụ**:

SELECT name, position, salaryFROM employeesWHERE salary > 4000;

**SELECT tất cả các cột**: Nếu muốn lấy tất cả các cột, bạn có thể sử dụng dấu \*.

SELECT \* FROM employees;

### 5. ****WHERE****

Câu lệnh **WHERE** được sử dụng để lọc kết quả theo điều kiện cụ thể.

SELECT column1, column2FROM table\_nameWHERE condition;

**Ví dụ**:

SELECT name, salaryFROM employeesWHERE salary > 4000;

### 6. ****JOIN****

**JOIN** được sử dụng để kết hợp dữ liệu từ hai hoặc nhiều bảng dựa trên một cột chung giữa các bảng.

#### a. ****INNER JOIN****: Chỉ trả về các bản ghi có giá trị tương ứng trong cả hai bảng.

SELECT column1, column2, ...FROM table1INNER JOIN table2ON table1.column = table2.column;

**Ví dụ**:

SELECT employees.name, departments.department\_nameFROM employeesINNER JOIN departmentsON employees.department\_id = departments.id;

#### b. ****LEFT JOIN****: Trả về tất cả các bản ghi từ bảng bên trái, và các bản ghi tương ứng từ bảng bên phải. Nếu không có bản ghi tương ứng, sẽ trả về NULL.

SELECT column1, column2, ...FROM table1LEFT JOIN table2ON table1.column = table2.column;

**Ví dụ**:

SELECT employees.name, departments.department\_nameFROM employeesLEFT JOIN departmentsON employees.department\_id = departments.id;

#### c. ****RIGHT JOIN****: Tương tự như ****LEFT JOIN****, nhưng trả về tất cả các bản ghi từ bảng bên phải và các bản ghi tương ứng từ bảng bên trái.

SELECT column1, column2, ...FROM table1RIGHT JOIN table2ON table1.column = table2.column;

### 7. ****GROUP BY****

Câu lệnh **GROUP BY** được sử dụng để gom nhóm dữ liệu dựa trên một hoặc nhiều cột và thường đi kèm với các hàm tổng hợp như **COUNT**, **SUM**, **AVG**, **MAX**, **MIN**.

SELECT column1, COUNT(\*)FROM table\_nameGROUP BY column1;

**Ví dụ**:

SELECT department\_id, COUNT(\*)FROM employeesGROUP BY department\_id;

Kết quả sẽ cho biết số lượng nhân viên trong mỗi phòng ban.

### 8. ****HAVING****

**HAVING** được sử dụng để lọc các nhóm dữ liệu sau khi đã được gom nhóm bằng **GROUP BY**. Nó giống với **WHERE**, nhưng hoạt động với các hàm tổng hợp.

SELECT column1, COUNT(\*)FROM table\_nameGROUP BY column1HAVING COUNT(\*) > value;

**Ví dụ**:

SELECT department\_id, COUNT(\*)FROM employeesGROUP BY department\_idHAVING COUNT(\*) > 10;

Kết quả trả về các phòng ban có hơn 10 nhân viên.

### 9. ****ORDER BY****

Câu lệnh **ORDER BY** được sử dụng để sắp xếp kết quả theo một hoặc nhiều cột. Mặc định là tăng dần (**ASC**), nếu muốn giảm dần, ta dùng **DESC**.

SELECT column1, column2FROM table\_nameORDER BY column1 ASC, column2 DESC;

**Ví dụ**:

SELECT name, salaryFROM employeesORDER BY salary DESC;

Kết quả sắp xếp nhân viên theo mức lương giảm dần.

### 10. ****LIMIT****

**LIMIT** được sử dụng để giới hạn số lượng bản ghi được trả về.

SELECT column1, column2FROM table\_name

LIMIT number\_of\_records;

**Ví dụ**:

SELECT name, salaryFROM employeesORDER BY salary DESC

LIMIT 5;

Kết quả sẽ trả về 5 nhân viên có mức lương cao nhất.

### 11. ****Alias (AS)****

**Alias** dùng để đặt bí danh cho cột hoặc bảng trong câu truy vấn, giúp câu lệnh dễ đọc hơn.

SELECT column1 AS alias\_nameFROM table\_name AS alias\_table;

**Ví dụ**:

SELECT e.name AS employee\_name, d.department\_nameFROM employees AS eJOIN departments AS dON e.department\_id = d.id;

### 12. ****Subquery****

Câu lệnh con (Subquery) là một câu lệnh **SELECT** lồng bên trong một câu lệnh SQL khác.

SELECT column1FROM table\_nameWHERE column2 = (SELECT column2 FROM other\_table WHERE condition);

**Ví dụ**:

SELECT nameFROM employeesWHERE salary > (SELECT AVG(salary) FROM employees);

Kết quả trả về các nhân viên có mức lương cao hơn mức lương trung bình.

VIII.**Docker** là một nền tảng phần mềm cho phép bạn tạo, triển khai, và chạy các ứng dụng bên trong các "container." Container là các gói phần mềm gọn nhẹ, có thể chạy đồng nhất trên bất kỳ môi trường nào, giúp phát triển và triển khai phần mềm dễ dàng và hiệu quả hơn.

### Các khái niệm chính trong Docker:

1. **Container**: Một đơn vị chạy ứng dụng đóng gói đầy đủ các phụ thuộc như mã nguồn, thư viện, và môi trường.
2. **Image**: Mẫu bất biến của container. Nó là snapshot của một ứng dụng với tất cả các phần mềm cần thiết để chạy container.
3. **Dockerfile**: Một tệp cấu hình được sử dụng để tạo image. Nó bao gồm các lệnh cần thiết để xây dựng ứng dụng và thiết lập môi trường bên trong container.
4. **Docker Hub**: Một kho lưu trữ trực tuyến, nơi bạn có thể tải lên hoặc tải về các Docker image.

### Lợi ích của Docker:

* **Tính di động**: Ứng dụng đóng gói trong container có thể chạy trên bất kỳ máy tính nào hỗ trợ Docker mà không cần quan tâm tới môi trường cài đặt.
* **Tối ưu hóa tài nguyên**: Docker container nhẹ hơn nhiều so với máy ảo (VM), vì chúng chia sẻ nhân hệ điều hành, giúp sử dụng tài nguyên hiệu quả hơn.
* **Tính nhất quán**: Môi trường phát triển, kiểm thử, và sản xuất có thể được đồng nhất.

### Các lệnh Docker cơ bản:

#### 1. ****Cài đặt Docker**** (Linux):

bash

sudo apt update

sudo apt install docker.io

#### 2. ****Kiểm tra Docker đã được cài đặt****

bash

docker --version

#### 3. ****Chạy container với image có sẵn****

Docker cung cấp một image có sẵn tên là hello-world để kiểm tra cài đặt:

bash

docker run hello-world

#### 4. ****Liệt kê các container đang chạy****

bash

docker ps

#### 5. ****Liệt kê tất cả các container (bao gồm cả container đã dừng)****

bash

docker ps -a

#### 6. ****Kéo (pull) một Docker image từ Docker Hub****

bash

docker pull ubuntu

#### 7. ****Tạo và chạy container từ image****

Bạn có thể tạo một container từ một image đã có:

bash

docker run -it ubuntu

Trong lệnh trên:

* -it: Gắn kết tương tác với terminal (interactive mode).
* ubuntu: Image của hệ điều hành Ubuntu.

#### 8. ****Chạy container trong nền (detached mode)****

bash

docker run -d nginx

Lệnh này chạy NGINX trong container và quay lại terminal ngay lập tức mà không hiển thị đầu ra.

#### 9. ****Dừng một container****

bash

docker stop <container\_id>

#### 10. ****Xóa container đã dừng****

bash

docker rm <container\_id>

#### 11. ****Xóa image****

bash

docker rmi <image\_id>

#### 12. ****Build Docker Image từ Dockerfile****

Tạo một tệp **Dockerfile** để định nghĩa cách tạo image. **Ví dụ Dockerfile**:

Dockerfile

# Sử dụng image nền từ Ubuntu

FROM ubuntu:20.04

# Cài đặt các gói cần thiết

RUN apt-get update && apt-get install -y python3

# Sao chép mã ứng dụng vào container

COPY . /app

# Thiết lập thư mục làm việc

WORKDIR /app

# Chạy ứng dụng Python khi container khởi động

CMD ["python3", "app.py"]

Chạy lệnh để build image từ Dockerfile:

bash

docker build -t my-python-app .

#### 13. ****Gắn volume để lưu dữ liệu****

Bạn có thể gắn volume để container có thể lưu dữ liệu ngoài:

bash

docker run -d -v /my/local/dir:/container/dir ubuntu

#### 14. ****Chạy container với cổng được mở ra ngoài****

bash

docker run -d -p 8080:80 nginx

Lệnh này ánh xạ cổng 80 trong container đến cổng 8080 của máy chủ.

### Docker Compose

**Docker Compose** cho phép bạn định nghĩa và chạy nhiều container trong một ứng dụng đa dịch vụ bằng cách sử dụng tệp YAML.

#### Ví dụ tệp docker-compose.yml:

yaml

version: '3'services:

web:

image: nginx

ports:

- "8080:80"

db:

image: mysql

environment:

MYSQL\_ROOT\_PASSWORD: password

Chạy tất cả các dịch vụ trong tệp docker-compose.yml:

docker-compose up