Spring Boot ITS

# Tổng quan về Spring Boot và cấu trúc dự án

1. Tìm hiểu về Java Spring Boot (tham khảo fptshop.com :v )
2. Spring Boot là gì ?

Spring Boot là một framework Java được sử dụng để xây dựng các ứng dụng và dịch vụ web dễ dàng và nhanh chóng. Nền tảng cung cấp các cấu hình mặc định cho một số thư viện và bộ công cụ hỗ trợ xây dựng, triển khai, quản lý ứng dụng Spring-based.

1. Ưu điểm nổi bật

* Tối ưu hóa quá trình phát triển: Spring Boot cung cấp cấu hình mặc định thông minh và tự động, giúp giảm thiểu việc cấu hình thủ công và tối ưu quá trình phát triển ứng dụng Java.
* Tích hợp tốt: Spring Boot tích hợp tốt với nhiều công nghệ và thư viện khác trong hệ sinh thái Spring Framework. Nền tảng cho phép hệ thống dễ dàng tích hợp các module và dịch vụ khác nhau mà không cần phải lo lắng về cấu hình phức tạp.
* Embedded server: Spring Boot đi kèm với các máy chủ nhúng như Tomcat, Jetty, hoặc Undertow. Đây là công cụ không thể thiếu trong việc triển khai ứng dụng một cách đơn giản mà không cần cấu hình thêm bất kỳ máy chủ nào khác.
* Tự động cấu hình: Spring Boot sử dụng cơ chế cấu hình tự động thông minh, cho phép ứng dụng tự cấu hình dựa trên các thư viện và module được sử dụng.
* Quản lý phụ thuộc: Spring Boot cung cấp các công cụ quản lý phụ thuộc mạnh mẽ như Maven hoặc Gradle, giúp quản lý các phụ thuộc của ứng dụng một cách hiệu quả.
* Monitoring và quản lý: Spring Boot cung cấp các công cụ hỗ trợ giám sát và quản lý ứng dụng dễ dàng, bao gồm Spring Boot Actuator cho việc giám sát và quản lý ứng dụng.

1. Thiết lập môi trường phát triển: IntelliJ IDEA, Mysql 8

* Sử dụng mail sinh viên đăng ký tài khoản dùng chùa :v
* Em cài maria trên docker sẵn từ trước rồi nên chắc skip đoạn này

1. Tạo và cấu hình dự án Spring Boot đầu tiên.
2. Create dự án bằng Spring initializr

* Vào <https://start.spring.io/> rồi select mấy cái option để tạo project theo ý muốn, tải về rồi giải nén thôi ;v

1. Create trên intellij

* new project rồi cũng select những option và dependency cần cho dự án

1. 1 vài dependency

* Lombok: nên chọn, nó giúp code Java ngắn hơn, nhưng cần cài thêm plugin Lombok vào IDE nữa
* Thymeleaf: chưa cần, Thymeleaf sẽ giúp pass data vào view của mô hình MVC, trả về trang HTML có data cho client
* Spring configuration processor, Spring devtools là các tool hỗ trợ thêm khi code thôi

1. Hiểu về cấu trúc dự án Spring Boot: các thư mục, file cấu hình, dependency management.
2. Cấu trúc chung

* Dù cho project được tạo với Maven hay Gradle thì cấu trúc chung vẫn tương tự nhau, do tuân theo một template có sẵn (tên là Archetype):
* src/main/java : chứa tất cả các lớp Java trong dự án. Thông thường, cấu trúc sẽ theo dạng các package để tổ chức mã nguồn.
* src/main/resources: Chứa các tài nguyên không phải mã nguồn (như cấu hình và file tĩnh).
* application.properties hoặc application.yml: File cấu hình chính của Spring Boot.
* static/: Chứa các tài nguyên tĩnh như HTML, CSS, JavaScript, hình ảnh, v.v.
* templates/: Chứa các template view (nếu sử dụng công cụ template engine như Thymeleaf, FreeMarker).
* META-INF/: Có thể chứa thông tin metadata.
* src/test/java: Chứa các lớp kiểm thử.
* pom.xml (với Maven) hoặc build.gradle (với Gradle): File cấu hình chính cho việc quản lý các phụ thuộc và cấu hình build của dự án.
* mvnw và mvnw.cmd: Script để chạy Maven Wrapper, cho phép build dự án mà không cần cài đặt Maven.
* Ví dụ:

my-spring-boot-project/

│

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── example/

│ │ │ └── demo/

│ │ │ ├── DemoApplication.java

│ │ │ ├── controller/

│ │ │ ├── service/

│ │ │ ├── repository/

│ │ │ └── model/

│ │ ├── resources/

│ │ ├── application.properties

│ │ ├── static/

│ │ ├── templates/

│ │ └── META-INF/

│ ├── test/

│ │ ├── java/

│ │ └── resources/

│

├── pom.xml

└── README.md

1. “Package theo layer”

* Controller layer: đặt trong controller, các class là controller sẽ có hậu tố Controller (ví dụ UserController, AuthController,...)
* Service layer: đặt trong service, các class có hậu tố là Service và thường tương ứng với controller (ví dụ UserService,...)
* Data access layer: ca này khó, bởi vì layer này bao gồm repository (đặt trong repository và hậu tố tương tự), DTO, model, entity... chi tiết mình sẽ nói ở các bài sau.
* util package chứa các lớp util (xử lý linh tinh), ví dụ như convert end date, tính toán đơn giản,...
* common package chứa các class định nghĩa như enum, interface, class dùng chung và đơn giản
* exception package chứa các class có nhiệm vụ xử lý exception trong Spring Boot.
* component chứa các bean được định nghĩa còn lại nhưng không thuộc layer nào.

:v hầu hết ứng dụng Spring Boot có tới 80% cấu trúc tương tự.

* Ví dụ minh hoạ

my-spring-boot-project/

│

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── example/

│ │ │ └── demo/

│ │ │ ├── DemoApplication.java

│ │ │ ├── controller/

│ │ │ │ └── UserController.java

│ │ │ │ └── AuthController.java

│ │ │ ├── service/

│ │ │ │ └── UserService.java

│ │ │ │ └── AuthService.java

│ │ │ ├── repository/

│ │ │ │ └── UserRepository.java

│ │ │ │ └── AuthRepository.java

│ │ │ ├── model/

│ │ │ │ └── User.java

│ │ │ │ └── Auth.java

│ │ │ ├── dto/

│ │ │ │ └── UserDTO.java

│ │ │ │ └── AuthDTO.java

│ │ │ ├── exception/

│ │ │ │ └── GlobalExceptionHandler.java

│ │ │ │ └── CustomException.java

│ │ │ ├── util/

│ │ │ │ └── DateUtil.java

│ │ │ │ └── CalculationUtil.java

│ │ │ ├── common/

│ │ │ │ └── Constants.java

│ │ │ │ └── AppEnums.java

│ │ │ ├── component/

│ │ │ │ └── JwtTokenProvider.java

│ │ ├── resources/

│ │ │ ├── application.properties

│ │ │ ├── static/

│ │ │ ├── templates/

│ │ │ └── META-INF/

│ ├── test/

│ │ ├── java/

│ │ └── resources/

│

├── pom.xml

└── README.md

1. “Package theo feature”

* Thay vì chia thành các package dựa theo layer, thì cách này chia theo tính năng. Nghĩa là mỗi tính năng, ví dụ user package thì sẽ chứa nào là UserController, UserService,...
* Ví dụ minh hoạ

my-spring-boot-project/

│

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── example/

│ │ │ └── demo/

│ │ │ ├── DemoApplication.java

│ │ │ ├── user/

│ │ │ │ ├── UserController.java

│ │ │ │ ├── UserService.java

│ │ │ │ ├── UserRepository.java

│ │ │ │ ├── UserDTO.java

│ │ │ │ └── User.java

│ │ │ ├── auth/

│ │ │ │ ├── AuthController.java

│ │ │ │ ├── AuthService.java

│ │ │ │ ├── AuthRepository.java

│ │ │ │ ├── AuthDTO.java

│ │ │ │ └── Auth.java

│ │ │ ├── exception/

│ │ │ │ ├── GlobalExceptionHandler.java

│ │ │ │ ├── CustomException.java

│ │ │ ├── util/

│ │ │ │ └── DateUtil.java

│ │ │ │ └── CalculationUtil.java

│ │ │ ├── common/

│ │ │ │ ├── Constants.java

│ │ │ │ └── AppEnums.java

│ │ │ ├── component/

│ │ │ │ └── JwtTokenProvider.java

│ │ ├── resources/

│ │ │ ├── application.properties

│ │ │ ├── static/

│ │ │ ├── templates/

│ │ │ └── META-INF/

│ ├── test/

│ │ ├── java/

│ │ └── resources/

│

├── pom.xml

└── README.md

1. file cấu hình application.properties

# Cấu hình cổng server

server.port=8081

# Cấu hình datasource (cơ sở dữ liệu)

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

spring.datasource.username=root

spring.datasource.password=pass123

# Hibernate JPA

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

# Cấu hình logging

logging.level.org.springframework=INFO

1. dependency management.

* Spring Boot cung cấp các **Starter** là các gói phụ thuộc đã được cấu hình sẵn để giúp bạn khởi động nhanh các thành phần của ứng dụng mà không cần phải tự thêm từng thư viện riêng lẻ.
* Ví dụ một số Spring Boot Starter phổ biến:
* **spring-boot-starter-web**: Dùng để phát triển các ứng dụng web, RESTful API.
* **spring-boot-starter-data-jpa**: Tích hợp với JPA và Hibernate để làm việc với cơ sở dữ liệu.
* **spring-boot-starter-security**: Cấu hình bảo mật cho ứng dụng.
* **spring-boot-starter-test**: Cung cấp các thư viện kiểm thử (JUnit, Mockito,...).
* **Ví dụ trong pom.xml (Maven)**:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

* **Ví dụ trong build.gradle (Gradle)**:

dependencies **{**

implementation 'org.springframework.boot:spring-boot-starter-web'

compileOnly 'org.projectlombok:lombok'

annotationProcessor 'org.springframework.boot:spring-boot-configuration-processor'

annotationProcessor 'org.projectlombok:lombok'

testImplementation 'org.springframework.boot:spring-boot-starter-test'

testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

**}**

1. Viết các controller đơn giản để xử lý request và response.
2. Controller !??

* Controller là nơi tiếp nhận request và trả về response cho client.
* Controller là một bean được đánh dấu với @Controller hoặc @RestController.

1. Các annotation trong spring boot

* @Controller Đánh dấu một class là một Controller trong Spring MVC. Controller này thường được dùng để xử lý các request và trả về view

@Controller

public class MyController {

// ...

}

* @RestController hoặc kết hợp giữa @Controller và @ResponseBody, dùng trong các ứng dụng RESTful, trả về dữ liệu trực tiếp (JSON hoặc XML) thay vì view. Khi sử dụng @RestController, tất cả các method trong class này đều mặc định trả về dữ liệu.

@RestController

public class MyRestController {

// ...

}

* @RequestMapping
* Được sử dụng để định nghĩa các đường dẫn và kiểu HTTP request (GET, POST, PUT, DELETE,...) cho cả class hoặc method.
* vd: localhost:8080/api/

@RequestMapping("/api")

public class MyController {

// ...

}

* Có thể áp dụng cho method để chỉ định cụ thể HTTP request và đường dẫn
* vd localhost:8080/users/…

@RequestMapping(value = "/users", method = RequestMethod.GET)

public List<User> getUsers() {

// ...

}

* @GetMapping Shortcut cho @RequestMapping(method = RequestMethod.GET), chỉ định xử lý HTTP GET request.

@GetMapping("/users")

public List<User> getAllUsers() {

// ...

}

* @PostMapping Shortcut cho @RequestMapping(method = RequestMethod.POST), chỉ định xử lý HTTP POST request.

@PostMapping("/users")

public User createUser(@RequestBody User user) {

// ...

}

* @PutMapping Shortcut cho @RequestMapping(method = RequestMethod.PUT), chỉ định xử lý HTTP PUT request.

@PutMapping("/users/{id}")

public User updateUser(@PathVariable Long id, @RequestBody User user) {

// ...

}

* @PatchMapping Shortcut cho @RequestMapping(method = RequestMethod.PATCH), dùng để cập nhật một phần tài nguyên.

@PatchMapping("/users/{id}")

public User partialUpdateUser(@PathVariable Long id, @RequestBody Map<String, Object> updates) {

// ...

}

* @DeleteMapping Shortcut cho @RequestMapping(method = RequestMethod.DELETE), chỉ định xử lý HTTP DELETE request.

@DeleteMapping("/users/{id}")

public void deleteUser(@PathVariable Long id) {

// ...

}

* @PathVariable Dùng để lấy dữ liệu từ URL.

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

public User getUserById(@PathVariable Long id) {

// ...

}

* @RequestParam Dùng để lấy dữ liệu từ query string (dòng tham số URL).

@GetMapping("/users")

public List<User> getUsers(@RequestParam String name) {

// ...

}

* @RequestBodyDùng để lấy dữ liệu từ body của HTTP request (thường dùng cho các method POST, PUT). trả về

@PostMapping("/users")

public User createUser(@RequestBody User user) {

// ...

}

* @RequestHeader Dùng để lấy thông tin từ HTTP header.

@GetMapping("/headers")

public String getHeaderInfo(@RequestHeader("User-Agent") String userAgent) {

return userAgent;

}

* @ResponseBody Dùng để trả về dữ liệu trực tiếp (thường là JSON, XML) thay vì trả về view. Với @RestController, annotation này được ngầm định.

@GetMapping("/text")

@ResponseBody

public String getText() {

return "Hello, World!";

}

* @ModelAttribute Dùng để gắn một object vào model hoặc lấy dữ liệu từ form HTML và tự động ánh xạ vào object.

@PostMapping("/users")

public String createUser(@ModelAttribute User user) {

// ...

}

* @CookieValue Dùng để lấy dữ liệu từ cookie.

@GetMapping("/cookie")

public String getCookieValue(@CookieValue("sessionId") String sessionId) {

return sessionId;

}

* @ResponseStatus Dùng để chỉ định mã trạng thái HTTP cho response.

@DeleteMapping("/users/{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void deleteUser(@PathVariable Long id) {

// ...

}

* @ExceptionHandler Dùng để xử lý các exception trong controller.

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

1. Ví dụ (lấy luôn trên trang chủ spring, )

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

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

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

@SpringBootApplication

@RestController

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.*run*(DemoApplication.class, args);

}

@GetMapping("/hello")

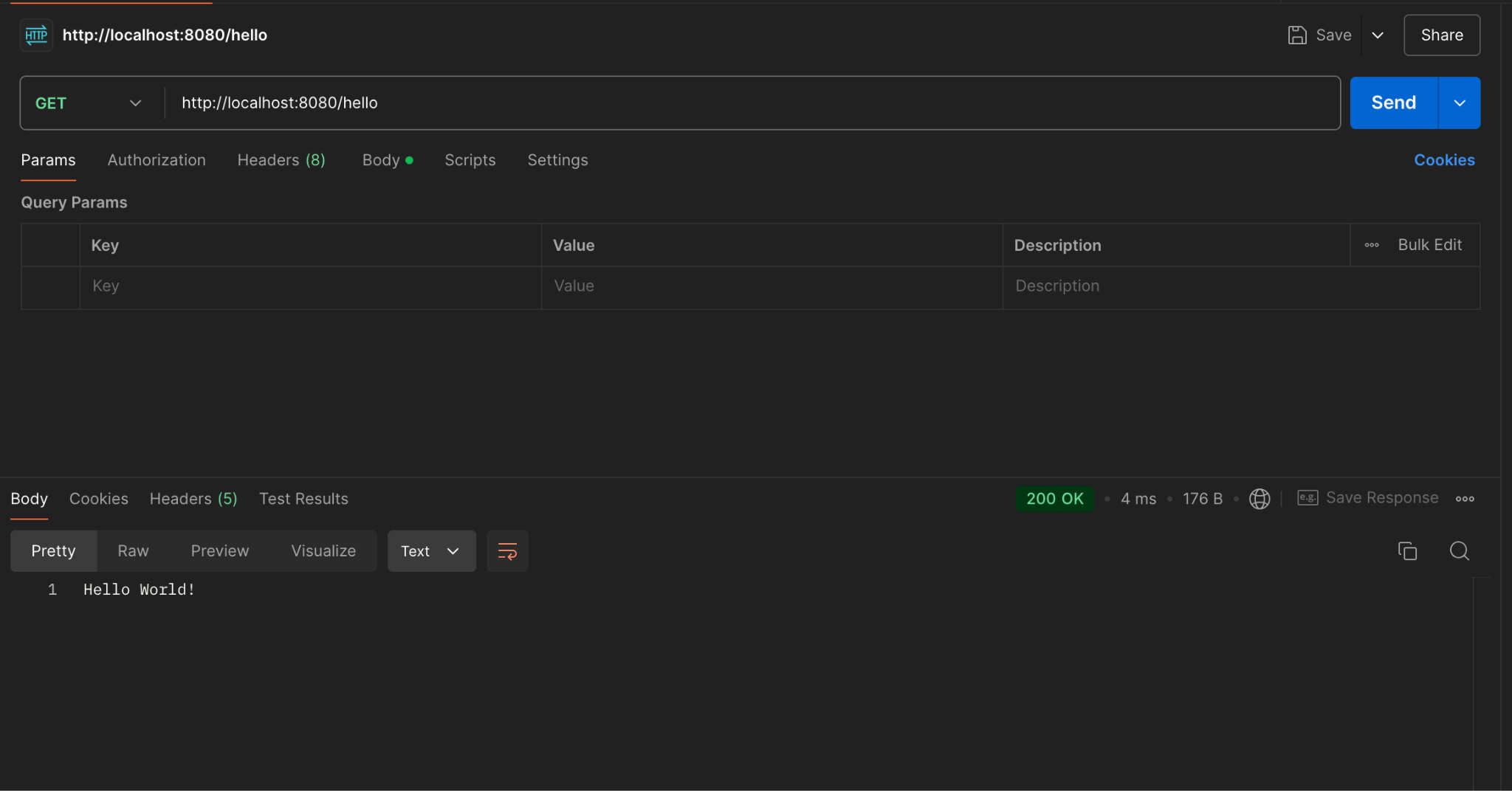
public String hello(@RequestParam(value = "name", defaultValue = "World") String name) {

return String.*format*("Hello %s!", name);

}

}

api trả ra :v



làm xong controller, rớt nước mắt

1. Cấu hình ứng dụng với application.properties
2. Cấu hình máy chủ (Server Configuration)

# Cấu hình cổng server

server.port=8081

# Địa chỉ để máy chủ lắng nghe

server.address=0.0.0.0

1. Cấu hình cơ sở dữ liệu (Database Configuration)

# Cấu hình kết nối cơ sở dữ liệu

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

spring.datasource.username=root

spring.datasource.password=123456

# Driver cơ sở dữ liệu

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

# Cấu hình JPA

spring.jpa.hibernate.ddl-auto=update # Hoặc create, validate, none

spring.jpa.show-sql=true # Hiển thị câu lệnh SQL trong console

spring.jpa.properties.hibernate.format\_sql=true # Định dạng câu SQL dễ đọc

1. Cấu hình Hibernate (JPA)

# Hibernate DDL auto: create, update, validate, none

spring.jpa.hibernate.ddl-auto=update

# Hiển thị các câu SQL đã thực hiện

spring.jpa.show-sql=true

# Định dạng câu lệnh SQL

spring.jpa.properties.hibernate.format\_sql=true

# Ngôn ngữ mặc định cho SQL

spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect

1. Cấu hình logging

# Cấu hình mức độ logging (error, warn, info, debug, trace)

logging.level.root=info

logging.level.org.springframework.web=debug

logging.level.com.example=trace

# Đường dẫn file log

logging.file.name=logs/myapp.log

1. Cấu hình thời gian session timeout

# Cấu hình thời gian chờ session

server.servlet.session.timeout=30m # 30 phút

1. Cấu hình email (Mail Configuration)

spring.mail.host=smtp.gmail.com

spring.mail.port=587

spring.mail.username=your-email@gmail.com

spring.mail.password=your-email-password

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

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

1. Cấu hình quốc tế hóa (Internationalization - i18n)

# Đường dẫn tới file chứa bản dịch

spring.messages.basename=messages

# Ngôn ngữ mặc định

spring.messages.default-locale=en

1. Cấu hình bảo mật (Security Configuration)

# Cấu hình bảo mật HTTP

spring.security.user.name=admin

spring.security.user.password=admin123

spring.security.user.roles=ADMIN,USER

1. Cấu hình upload file

# Cấu hình giới hạn kích thước tệp tải lên

spring.servlet.multipart.max-file-size=5MB

spring.servlet.multipart.max-request-size=5MB

1. Cấu hình cache

# Bật cache

spring.cache.type=*simple*

# Cache với thời gian hết hạn

spring.cache.cache-names=cache1, cache2

spring.cache.cache1.time-to-live=600000 # 10 phút

1. Cấu hình Thymeleaf

# Cấu hình Thymeleaf

spring.thymeleaf.cache=false

spring.thymeleaf.mode=HTML5

spring.thymeleaf.encoding=UTF-8

spring.thymeleaf.prefix=classpath:/templates/

spring.thymeleaf.suffix=.html

1. Cấu hình Spring Boot Actuator

# Bật tất cả các endpoint Actuator

management.endpoints.web.exposure.include=\*

# Bật các endpoint cụ thể (health, info)

management.endpoint.health.show-details=always

management.endpoint.health.enabled=true

management.endpoint.info.enabled=true

1. Cấu hình profile (Môi trường cấu hình khác nhau)

# Cấu hình profile

spring.profiles.active=dev

1. Cấu hình Web (Servlet, Tomcat)
2. # Thay đổi kích thước thread pool của Tomcat
3. server.tomcat.threads.max=200
4. server.tomcat.threads.min-spare=10
5. Nghiên cứu các câu lệnh đơn giản về truy vấn Mysql (Insert, update, select, join, group,...)
6. Tạo cơ sở dữ liệu (CREATE DATABASE):

CREATE DATABASE ten\_csdl;

1. Xóa cơ sở dữ liệu (DROP DATABASE):

DROP DATABASE ten\_csdl;

1. Sửa thông tin cơ sở dữ liệu (ALTER DATABASE):

ALTER DATABASE ten\_csdl COLLATE collation;

1. Tạo bảng (CREATE TABLE):

CREATE TABLE Student(

ID INT AUTO\_INCREMENT PRIMARY KEY,

FullName VARCHAR(255),

Age INT DEFAULT 22,

City VARCHAR(255) DEFAULT 'Cần Thơ'

) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci;

* auto\_increment cho thằng khoá kiểu int để auto tăng gtri
* ko biết sao nhưng nó bị lỗi 1067 (42000): Invalid default value, so tra set unicode cho nó mới set default có dấu tiếng việt được

1. Xóa bảng (DROP TABLE):

DROP TABLE ten\_bang;

1. Thêm cột (ALTER TABLE..ADD):

ALTER TABLE ten\_bang ADD ten\_cot KIEU\_DU\_LIEU;

1. Xóa cột (ALTER TABLE..DROP COLUMN):

ALTER TABLE ten\_bang DROP COLUMN ten\_cot;

1. Sửa cột để không nhận giá trị null (ALTER TABLE..ALTER COLUMN):

ALTER TABLE ten\_bang MODIFY ten\_cot KIEU\_DU\_LIEU NOT NULL;

Ví dụ:

ALTER TABLE students MODIFY name VARCHAR(100) NOT NULL;

1. Thêm khóa chính (ALTER TABLE..ADD PRIMARY KEY):

ALTER TABLE ten\_bang ADD PRIMARY KEY (ten\_cot);

1. Thêm constraint khóa ngoài (ALTER TABLE..ADD FOREIGN KEY):

ALTER TABLE ten\_bang ADD CONSTRAINT fk\_ten FOREIGN KEY (ten\_cot) REFERENCES ten\_bang\_khac(ten\_cot\_khac);

1. Thêm ràng buộc mặc định (ALTER TABLE..ADD DEFAULT):

ALTER TABLE ten\_bang ALTER ten\_cot SET DEFAULT gia\_tri\_mac\_dinh;

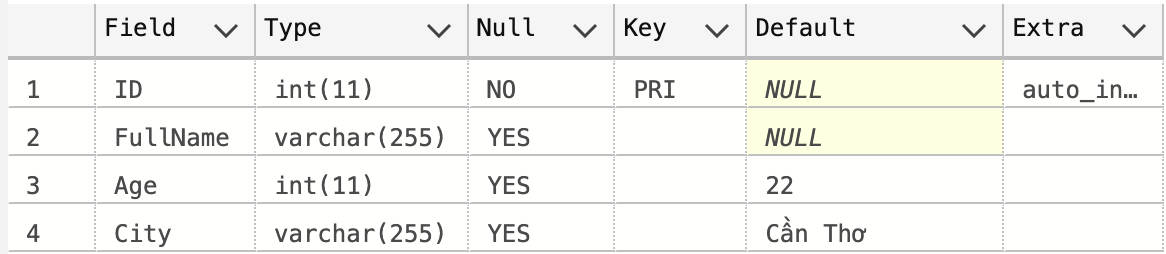
Ví dụ:

ALTER TABLE students ALTER age SET DEFAULT 18;

1. Xem thông tin bảng

DESCRIBE ten\_bang;

Ví dụ output



1. Thêm bản ghi (INSERT..VALUES):

INSERT INTO ten\_bang (cot1, cot2, ...) VALUES (giatri1, giatri2, ...);

1. Thêm bản ghi từ bảng khác (INSERT..SELECT):

INSERT INTO ten\_bang1 (cot1, cot2, ...) SELECT cot1, cot2, ... FROM ten\_bang2;

Ví dụ

INSERT INTO graduates (id, name) SELECT id, name FROM students WHERE age > 22;

1. Tạo bảng mới từ kết quả chọn (SELECT..INTO):

CREATE TABLE ten\_bang\_moi AS SELECT cot1, cot2 FROM ten\_bang;

1. Xóa bản ghi (DELETE):

DELETE FROM ten\_bang WHERE dieu\_kien;

1. Xóa toàn bộ bản ghi (TRUNCATE):

TRUNCATE TABLE ten\_bang;

1. Sửa bản ghi (UPDATE):

UPDATE ten\_bang SET cot1 = giatri\_moi WHERE dieu\_kien;

1. Hiển thị tất cả (SELECT \*):

SELECT \* FROM ten\_bang;

1. Hiển thị một số cột (SELECT):

SELECT cot1, cot2 FROM ten\_bang;

1. Lọc dữ liệu (SELECT..WHERE):

SELECT \* FROM ten\_bang WHERE dieu\_kien;

1. Kiểm tra giá trị có nằm trong danh sách (IN)

SELECT column1, column2 FROM table\_name WHERE column1 IN (value1, value2, ...);

Ví dụ:

SELECT first\_name, last\_name FROM employees WHERE department\_id IN (1, 2, 3);

1. Kiểm tra giá trị trong khoảng (BETWEEN)

SELECT column1, column2 FROM table\_name WHERE column1 BETWEEN value1 AND value2;

1. Tìm kiếm theo mẫu (LIKE)

SELECT column1, column2 FROM table\_name WHERE column1 LIKE pattern;

* **%**: Đại diện cho bất kỳ chuỗi ký tự nào (bao gồm cả chuỗi rỗng).
* 'J%' sẽ khớp với bất kỳ chuỗi nào bắt đầu bằng chữ "J", ví dụ: "John", "Jane", "Jake".
* %a% có nghĩa là bất kỳ chuỗi nào có chứa chữ "a" ở bất kỳ vị trí nào, ví dụ: "Jane", "Sara", "Amanda".
* %j sẽ khớp với tất cả các chuỗi kết thúc bằng chữ "j".
* **\_**: Đại diện cho một ký tự duy nhất.
* \_ đại diện cho một ký tự duy nhất. Do đó, mẫu 'J\_\_\_' sẽ tìm các tên có đúng 4 ký tự và bắt đầu bằng chữ "J", ví dụ: "John", "Jack".
* \_o% có nghĩa là tên phải có chữ "o" là ký tự thứ hai và có thể có bất kỳ chuỗi ký tự nào sau đó, ví dụ: "John", "Tom", "Bob".
* **NOT LIKE** được sử dụng để tìm các giá trị không khớp với mẫu.

Ví dụ:

SELECT first\_name, last\_name FROM employees WHERE first\_name LIKE 'J%';

SELECT first\_name

FROM employees

WHERE first\_name LIKE 'J\_\_\_';

1. Sắp xếp dữ liệu (SELECT..ORDER BY):

SELECT \* FROM ten\_bang ORDER BY cot1 ASC|DESC;

1. Hiển thị giá trị không trùng lặp (SELECT..DISTINCT):

SELECT DISTINCT cot1 FROM ten\_bang;

1. Lấy các dòng đầu tiên (SELECT..TOP):

SELECT \* FROM ten\_bang LIMIT so\_dong;

Ví dụ:

SELECT \* FROM students LIMIT 5;

1. Lồng truy vấn (SELECT..WHERE (SELECT)):

SELECT \* FROM ten\_bang WHERE cot IN (SELECT cot\_khac FROM bang\_khac);

Ví dụ:

SELECT \* FROM students WHERE class\_id IN (SELECT id FROM classes WHERE name = 'Math');

Ví dụ phát nữa

SELECT first\_name, last\_name,

(SELECT department\_name

FROM departments

WHERE departments.id = employees.department\_id) AS department\_name

FROM employees;

1. Nhóm các bản ghi (GROUP BY)

SELECT column1, FROM table\_name GROUP BY column1;

1. Nhóm kết quả (SELECT..HAVING):

SELECT cot1, … FROM ten\_bang GROUP BY cot1, … HAVING condition;

Ví dụ:

SELECT age, COUNT(\*) FROM Student GROUP BY age HAVING COUNT(\*) > 1;

1. Nối bảng (SELECT..INNER JOIN):

SELECT cot1, cot2 FROM bang1 INNER JOIN bang2 ON bang1.cot\_chung = bang2.cot\_chung;

Ví dụ:

SELECT students.name, classes.name FROM students INNER JOIN classes ON students.class\_id = classes.id;

1. Đặt tên khác cho cột hoặc bảng (ALIAS)

SELECT column1 AS alias\_name

FROM table\_name AS alias\_table;

Ví dụ:

SELECT first\_name AS fname, last\_name AS lname

FROM employees AS e;

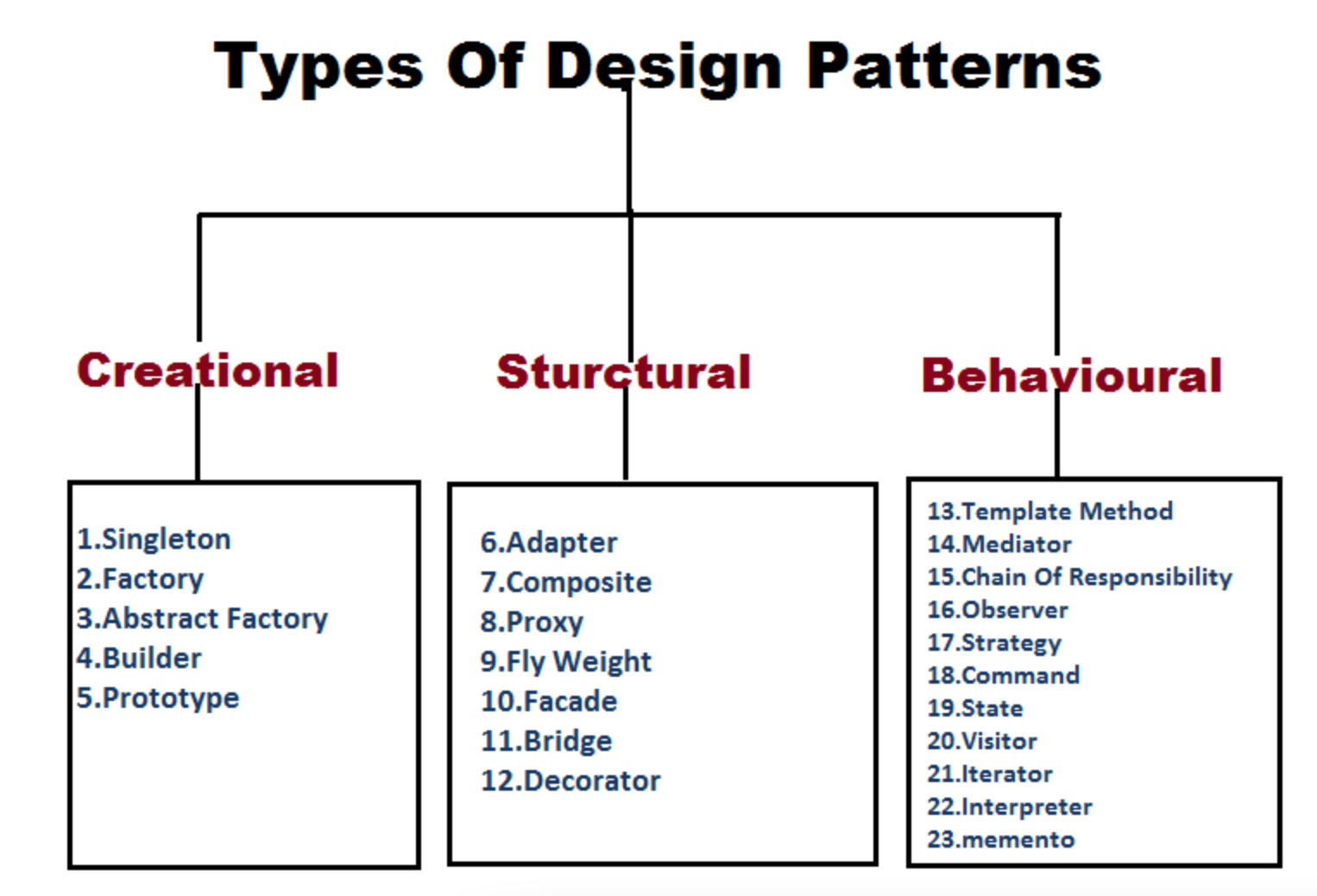
1. Kết hợp kết quả của nhiều truy vấn (UNION)

SELECT column1, column2 FROM table1

UNION

SELECT column1, column2 FROM table2;

1. gg
2. Tìm hiểu các design pattern cơ bản trong java

* Các design pattern trong java thường được chia làm 3 nhóm
* 1 số design parttern khác
* [Dependency Injection](https://stackjava.com/design-pattern/dependency-injection-di-la-gi.html)
* Thread Safety in Java Singleton
* So sánh các nhóm design pattern

| 1 | **Creational Patterns** | Mẫu thiết kế này cung cấp phương pháp tạo ra các đối tượng một cách linh hoạt hơn. Nghĩa là quyết định đối tượng nào được tạo ra tuỳ thuộc vào trường hợp sử dụng nhất định. |
| --- | --- | --- |
| 2 | **Structural Patterns** | Mẫu thiết kế liên quan đến sự kết hợp giữa các đối tượng với nhau |
| 3 | **Behavioral Patterns** | Mẫu thiết kế này trình bày phương pháp thiết kế liên quan đến hành vi của các đối tượng (objects) |

# 

# Tìm hiểu Design pattern

1. Mục đích các design pattern

* creational pattern
* **Singleton Pattern**: Đảm bảo rằng một lớp chỉ có một thể hiện duy nhất và cung cấp một điểm truy cập toàn cầu cho nó. Thường được sử dụng cho các lớp quản lý tài nguyên, như cấu hình hoặc kết nối cơ sở dữ liệu.
* Factory Method Pattern: Định nghĩa một interface cho việc tạo đối tượng, nhưng để cho các lớp con quyết định lớp nào sẽ được khởi tạo. Giúp tạo ra đối tượng mà không cần phải chỉ định lớp cụ thể.
* Abstract Factory Pattern: Cung cấp một interface để tạo ra một họ các đối tượng mà không cần chỉ định lớp cụ thể. Thường được sử dụng khi một hệ thống cần phải độc lập với cách tạo ra, cấu hình và biểu diễn của các đối tượng.
* Builder Pattern: Tách quá trình tạo một đối tượng phức tạp thành các bước riêng biệt. Cho phép tạo ra các đối tượng khác nhau với cùng một quy trình xây dựng
* Prototype Pattern: Sử dụng một thể hiện có sẵn để tạo ra các đối tượng mới thông qua sao chép. Thích hợp khi việc tạo ra đối tượng mới tốn kém về tài nguyên hoặc thời gian.
* Structural pattern
* Adapter Pattern: Cho phép các lớp không tương thích làm việc cùng nhau bằng cách chuyển đổi interface của một lớp thành interface mà các lớp khác mong đợi. Thích hợp khi muốn kết hợp các hệ thống khác nhau.
* Bridge Pattern: Tách biệt abstraction (trừu tượng) và implementation (thực thi) để chúng có thể phát triển độc lập. Giúp giảm sự phụ thuộc giữa các lớp và cải thiện tính linh hoạt.
* Composite Pattern: Cho phép bạn tạo ra một cấu trúc cây để biểu diễn các đối tượng đơn lẻ và nhóm các đối tượng. Hỗ trợ việc xử lý đồng nhất giữa các đối tượng đơn và nhóm
* Decorator Pattern: Cho phép thêm các chức năng mới cho một đối tượng hiện có mà không thay đổi cấu trúc của nó. Hỗ trợ việc mở rộng các đối tượng một cách linh hoạt.
* Facade Pattern: Cung cấp một interface đơn giản cho một tập hợp các interface phức tạp trong một hệ thống. Giúp giảm sự phụ thuộc và làm cho hệ thống dễ sử dụng hơn.
* Flyweight Pattern: Giúp tiết kiệm bộ nhớ bằng cách chia sẻ các đối tượng tương tự. Thích hợp khi có rất nhiều đối tượng giống nhau trong một ứng dụng.
* Proxy Pattern: Cung cấp một đại diện hoặc một đối tượng thay thế cho một đối tượng khác. Có thể được sử dụng để kiểm soát quyền truy cập, lazy loading hoặc ghi lại hoạt động.
* Behavioral pattern
* Chain of Responsibility Pattern: Cho phép nhiều đối tượng xử lý một yêu cầu mà không cần phải biết đối tượng nào sẽ xử lý yêu cầu đó. Mỗi đối tượng có thể truyền yêu cầu đến đối tượng tiếp theo trong chuỗi.
* Command Pattern: Biến yêu cầu thành một đối tượng, cho phép lưu trữ, lên hàng đợi hoặc ghi lại các yêu cầu. Hỗ trợ việc thực hiện các hành động mà không cần biết cách thực thi chúng.
* Interpreter Pattern: Cung cấp một cách để định nghĩa ngữ nghĩa cho một ngôn ngữ hoặc một cú pháp thông qua các lớp. Thích hợp cho việc xử lý cú pháp và ngữ nghĩa của các ngôn ngữ.
* Iterator Pattern: Cung cấp một cách để truy cập các phần tử của một tập hợp mà không cần tiết lộ cấu trúc bên trong của nó. Hỗ trợ việc duyệt qua các đối tượng trong một cấu trúc dữ liệu.
* Mediator Pattern: Giảm thiểu sự phụ thuộc giữa các đối tượng bằng cách sử dụng một đối tượng trung gian (mediator) để giao tiếp. Giúp làm cho hệ thống dễ bảo trì và mở rộng hơn.
* Memento Pattern: Cho phép lưu trữ và khôi phục trạng thái của một đối tượng mà không làm lộ cấu trúc bên trong của nó. Thích hợp cho việc thực hiện tính năng undo/redo
* Observer Pattern: Xác định một mối quan hệ một-nhiều giữa các đối tượng, sao cho khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc được thông báo và cập nhật tự động.
* State Pattern: Cho phép một đối tượng thay đổi hành vi của nó khi trạng thái của nó thay đổi. Hỗ trợ việc quản lý các trạng thái phức tạp mà không cần sử dụng nhiều cấu trúc điều kiện.
* Strategy Pattern: Định nghĩa một họ các thuật toán, đóng gói chúng lại và cho phép chúng có thể thay đổi độc lập với các đối tượng sử dụng chúng. Hỗ trợ việc thay đổi thuật toán trong thời gian chạy.
* Template Method Pattern: Định nghĩa một thuật toán trong một phương thức, nhưng cho phép các lớp con triển khai một số bước của thuật toán. Giúp tái sử dụng mã và giảm sự trùng lặp.

# Vì ko thể tìm hiểu hết đống kia nên e tìm hiểu 3 cái sau (thấy các tiền bối gthieu)

1. Singleton pattern

* Chức năng: Singleton đảm bảo chỉ duy nhất môt new instance được tạo ra và nó sẽ cung cấp cho bạn một method để truy cập đến thực thể đó.
* Trường hợp sử dụng:
* Lớp ghi nhật ký (Logger).
* Lớp cấu hình (Configuration).
* Các nhóm kết nối cơ sở dữ liệu (Database connection pools).
* Nguyên tắc: Dù cho việc thực hiện bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây.
* private constructor để hạn chế truy cập từ class bên ngoài
* đặt private static variable đảm bảo biến chỉ được khởi tạo trong class.
* có một method public để return instance được khởi tạo ở trên.
* Ưu điểm
* Kiểm soát truy cập đến thể hiện duy nhất.
* Giảm thiểu lượng bộ nhớ so với việc tạo nhiều thể hiện.
* Nhược điểm
* Có thể giới thiệu trạng thái toàn cục vào ứng dụng, điều này có thể dẫn đến các vấn đề với việc kiểm thử và xử lý đồng thời.
* Không thích hợp cho các môi trường đa luồng trừ khi được đồng bộ hóa đúng cách.
* Có một số cách để triển khai mẫu Singleton:
* Khởi tạo sớm (Eager Initialization):

public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() { }

public static Singleton getInstance() {

return instance;

}

}

* Khởi tạo muộn (Lazy Initialization):

public class Singleton {

private static Singleton instance;

private Singleton() { }

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

* Khởi tạo muộn an toàn với luồng (Thread-Safe Lazy Initialization):

public class Singleton {

private static Singleton instance;

private Singleton() { }

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

* Triển khai Bill Pugh (Sử dụng lớp tĩnh nội bộ):

public class Singleton {

private Singleton() { }

private static class SingletonHelper {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHelper.INSTANCE;

}

}

1. Adapter pattern

* Mục tiêu chính của Adapter là cho phép một đối tượng với một giao diện cụ thể (giao diện không tương thích) có thể sử dụng được trong một hệ thống yêu cầu một giao diện khác.
* Trường hợp sử dụng
* Khi muốn sử dụng lại các lớp hiện có nhưng giao diện của chúng không phù hợp với yêu cầu.
* Khi cần hợp nhất các hệ thống hoặc lớp có giao diện không tương thích.
* Các thành phần của Adapter Pattern
* Target (interface) là giao diện mà client mong muốn. Nó là định dạng chung mà client sử dụng để thực hiện các hành động.
* Client là lớp hoặc phần trong hệ thống cần gọi một giao diện để thực hiện công việc. Tuy nhiên, giao diện mà client cần không khớp với giao diện mà một lớp cụ thể cung cấp.
* Adaptee là lớp với giao diện không tương thích với client. Tuy nhiên, Adaptee lại có những chức năng mà client cần.
* Adapter là lớp trung gian chịu trách nhiệm chuyển đổi các lời gọi từ client sang dạng mà lớp Adaptee có thể hiểu và xử lý. Adapter triển khai giao diện Target và bên trong nó, chuyển đổi các lệnh từ client sang lệnh mà Adaptee sử dụng.
* Ưu điểm
* **Tăng tính tái sử dụng:** Bạn có thể sử dụng lại các lớp đã tồn tại với giao diện khác mà không cần thay đổi mã nguồn của chúng.
* **Giảm sự phức tạp:** Adapter giúp đơn giản hóa việc tích hợp các hệ thống có giao diện khác nhau.
* Triển khai
* Giả sử có một giao diện MediaPlayer phát nhạc và một lớp AdvancedMediaPlayer phát định dạng âm thanh cao cấp như MP4 và VLC. Adapter sẽ giúp MediaPlayer sử dụng AdvancedMediaPlayer.
* Giao diện Target (Client yêu cầu)

public interface MediaPlayer {

void play(String audioType, String fileName);

}

* Adaptee (Lớp có giao diện không tương thích)

public class AdvancedMediaPlayer {

public void playMp4(String fileName) {

System.out.println("Playing MP4 file. Name: " + fileName);

}

}

* Lớp Adapter

public class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter() {

advancedMusicPlayer = new AdvancedMediaPlayer();

}

@Override

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer.playMp4(fileName);

}

}

}

* Lớp Client

public class AudioPlayer {

private MediaPlayer mediaPlayer;

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("mp4")) {

mediaPlayer = new MediaAdapter();

mediaPlayer.play(audioType, fileName);

} else {

System.out.println("Unsupported format: " + audioType);

}

}

}

* Gọi client

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp4", "song.mp4");

}

}

* loading ♾️

1. Chain of responsibility

* Mục tiêu chính của mẫu thiết kế Chain of Responsibility là **tách riêng việc gửi yêu cầu khỏi các đối tượng xử lý yêu cầu**, bằng cách thiết lập một chuỗi các đối tượng. Điều này giúp hệ thống linh hoạt hơn, dễ dàng mở rộng khi thêm các đối tượng xử lý mà không cần thay đổi client.
* Trường hợp sử dụng
* Khi bạn muốn **gửi yêu cầu đến nhiều đối tượng** mà không cần xác định trước đối tượng nào sẽ xử lý yêu cầu.
* Khi có nhiều điều kiện xử lý khác nhau, và cần một cơ chế linh hoạt để **chuyển tiếp yêu cầu** qua các điều kiện đó.
* Các thành phần chính của Chain of Responsibility
* Handler (Xử lý yêu cầu) là giao diện hoặc lớp trừu tượng (abstract class) mà tất cả các đối tượng xử lý phải triển khai hoặc kế thừa. Nó định nghĩa phương thức để xử lý yêu cầu, đồng thời chứa một tham chiếu đến đối tượng kế tiếp trong chuỗi.
* ConcreteHandler (xử lý cụ thể) các lớp cụ thể sẽ kế thừa hoặc triển khai Handler. Mỗi lớp này sẽ xử lý một yêu cầu cụ thể và quyết định có cần chuyển tiếp yêu cầu đến handler kế tiếp trong chuỗi hay không.
* Client: gửi yêu cầu tới đối tượng đầu tiên trong chuỗi và không cần biết đối tượng nào sẽ xử lý yêu cầu.
* Ưu:
* **Giảm sự phụ thuộc giữa client và các lớp xử lý cụ thể:** Client không cần biết đối tượng nào sẽ xử lý yêu cầu, mà chỉ cần gửi yêu cầu đến chuỗi.
* **Dễ dàng mở rộng và bảo trì:** Bạn có thể thêm hoặc thay đổi các đối tượng xử lý mà không ảnh hưởng đến client.
* **Linh hoạt trong việc xử lý yêu cầu:** Cho phép nhiều đối tượng có cơ hội xử lý yêu cầu.
* Nhược:
* **Khó theo dõi luồng xử lý:** Nếu chuỗi quá dài hoặc phức tạp, rất khó để xác định đối tượng nào đã xử lý yêu cầu.
* **Không đảm bảo rằng yêu cầu sẽ được xử lý:** Nếu không có đối tượng nào trong chuỗi xử lý yêu cầu, yêu cầu có thể bị bỏ qua.
* Đề mô:
* Giả sử có một hệ thống xử lý yêu cầu cấp quyền với các cấp độ: Người Quản lý, Giám đốc, và CEO. Mỗi cấp độ xử lý yêu cầu khác nhau, và nếu một cấp độ không xử lý được, yêu cầu sẽ được chuyển đến cấp độ cao hơn.
* Lớp Handler

public abstract class Approver {

protected Approver nextApprover;

public void setNextApprover(Approver nextApprover) {

this.nextApprover = nextApprover;

}

public abstract void approveRequest(int amount);

}

* Các lớp cụ thể ConcreteHandler

public class Manager extends Approver {

@Override

public void approveRequest(int amount) {

if (amount <= 1000) {

System.out.println("Manager approves request of $" + amount);

} else if (nextApprover != null) {

nextApprover.approveRequest(amount);

}

}

}

public class Director extends Approver {

@Override

public void approveRequest(int amount) {

if (amount <= 5000) {

System.out.println("Director approves request of $" + amount);

} else if (nextApprover != null) {

nextApprover.approveRequest(amount);

}

}

}

public class CEO extends Approver {

@Override

public void approveRequest(int amount) {

if (amount > 5000) {

System.out.println("CEO approves request of $" + amount);

}

}

}

* Tạo lớp Client và thiết lập chuỗi

public class ChainOfResponsibilityDemo {

public static void main(String[] args) {

// Tạo các đối tượng cụ thể

Approver manager = new Manager();

Approver director = new Director();

Approver ceo = new CEO();

// Thiết lập chuỗi xử lý

manager.setNextApprover(director);

director.setNextApprover(ceo);

// Client gửi yêu cầu

manager.approveRequest(500);// Manager approves

manager.approveRequest(2000);// Director approves

manager.approveRequest(7000);// CEO approves

}

}

* loading ♾️
* loading ♾️

1. loading ♾️
2. Hiểu cơ bản về Dependency Injection (DI) và Inversion of Control (IoC).
3. Trước tiên. Khái niệm tight-coupling và cách loosely coupled
4. tight-coupling (liên kết ràng buộc)

* Thuật ngữ ám chỉ mối quan hệ giữa các lớp (classes) quá chặt chẽ. Khi sử dụng tight-coupling, các lớp kết nối với nhau một cách mạnh mẽ, và sự thay đổi trong một lớp có thể ảnh hưởng đến toàn bộ hệ thống hoặc các lớp khác. Điều này có thể tạo ra sự phụ thuộc không mong muốn và làm cho mã nguồn trở nên khó bảo trì và mở rộng.
* Ví dụ:
* Giả sử có một lớp VeryComplexService thực hiện một nhiệm vụ phức tạp, trong đó một phần của nhiệm vụ là sắp xếp một mảng dữ liệu trước khi xử lý

public class BubbleSortAlgorithm {

public void sort(int[] array) {

// TODO: Add your sorting logic here

System.out.println("Sorted using bubble sort algorithm");

}

}

public class VeryComplexService {

private BubbleSortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

public VeryComplexService() {

}

public void complexBusiness(int[] array) {

bubbleSortAlgorithm.sort(array);

// TODO: More business logic here

}

}

* VeryComplexService và BubbleSortAlgorithm quá chặt chẽ kết nối với nhau. Khi bạn cần thay đổi thuật toán sắp xếp sang QuickSort, bạn sẽ phải sửa cả hai lớp.
* loading ♾️

1. loose coupling (liên kết lỏng)

* Là cách để giảm bớt sự phụ thuộc giữa các lớp với nhau. Trong loose coupling, các lớp hoạt động độc lập và không biết gì về cấu trúc hoặc chi tiết triển khai của các lớp khác. Điều này tạo điều kiện thuận lợi cho việc mở rộng và bảo trì code
* Ví dụ
* Sử dụng một giao diện (interface) SortAlgorithm để giảm sự phụ thuộc giữa các lớp

public interface SortAlgorithm {

void sort(int[] array);

}

// Thuật toán sắp xếp BubbleSort

public class BubbleSort implements SortAlgorithm {

@Override

public void sort(int[] array) {

// TODO: Add bubble sort logic here

System.out.println("Sorted using BubbleSort");

}

}

// Thuật toán sắp xếp QuickSort

public class QuickSort implements SortAlgorithm {

@Override

public void sort(int[] array) {

// TODO: Add quick sort logic here

System.out.println("Sorted using QuickSort");

}

}

public class VeryComplexService {

private SortAlgorithm sortAlgorithm;

// Inject thuật toán sắp xếp thông qua constructor

public VeryComplexService(SortAlgorithm sortAlgorithm) {

this.sortAlgorithm = sortAlgorithm;

}

// Một phương thức phức tạp sử dụng thuật toán sắp xếp

public void performComplexOperation(int[] data) {

System.out.println("Performing complex operation...");

sortAlgorithm.sort(data);

System.out.println("Complex operation finished.");

}

}

public class Main {

public static void main(String[] args) {

int[] data = { 5, 2, 9, 1, 5, 6 };

// Sử dụng BubbleSort

SortAlgorithm bubbleSort = new BubbleSort();

VeryComplexService service1 = new VeryComplexService(bubbleSort);

service1.performComplexOperation(data);

// Sử dụng QuickSort

SortAlgorithm quickSort = new QuickSort();

VeryComplexService service2 = new VeryComplexService(quickSort);

service2.performComplexOperation(data);

}

}

* VeryComplexService không còn phụ thuộc vào cụ thể của thuật toán sắp xếp. Bạn có thể dễ dàng thay đổi thuật toán bằng cách truyền một đối tượng thuật toán khác vào VeryComplexService.
* loading ♾️

1. loading ♾️
2. Dependency Injection
3. Sau khi có khái niệm về loose coupling thì :v

* Dependency Injection (DI) là một cách để thực hiện loose coupling , trong đó các phụ thuộc của một lớp không được tạo bên trong lớp đó mà được cung cấp từ bên ngoài. Điều này giúp tách biệt việc xây dựng và cấu hình đối tượng từ việc sử dụng nó. Trong ví dụ trên, VeryComplexService sử dụng DI để nhận một đối tượng SortAlgorithm từ bên ngoài.

=> túm lại, **DI** cung cấp các **phụ thuộc** (dependencies) cho một đối tượng từ bên ngoài, thay vì để đối tượng đó tự tạo ra phụ thuộc của chính nó. Các phụ thuộc thường là những đối tượng mà đối tượng hiện tại cần sử dụng để hoàn thành công việc của nó.

1. Các hình thức của DI

* **Constructor Injection:** Các phụ thuộc được tiêm vào đối tượng thông qua constructor. Khi khởi tạo một đối tượng, ta cung cấp các phụ thuộc ngay từ ban đầu thông qua constructor.

public class Service {

private Dependency dependency;

// Constructor Injection

public Service(Dependency dependency) {

this.dependency = dependency;

}

public void perform() {

dependency.execute();

}

}

* **Setter Injection:** Các phụ thuộc được cung cấp cho đối tượng thông qua các phương thức setter. Điều này cho phép bạn khởi tạo đối tượng trước rồi mới "tiêm" các phụ thuộc sau

public class Service {

private Dependency dependency;

// Setter Injection

public void setDependency(Dependency dependency) {

this.dependency = dependency;

}

public void perform() {

dependency.execute();

}

}

* **Interface Injection:** Các phụ thuộc được tiêm vào thông qua các phương thức của interface mà đối tượng phải triển khai. Cách này ít được sử dụng hơn.

public interface DependencyAware {

void setDependency(Dependency dependency);

}

public class Service implements DependencyAware {

private Dependency dependency;

@Override

public void setDependency(Dependency dependency) {

this.dependency = dependency;

}

public void perform() {

dependency.execute();

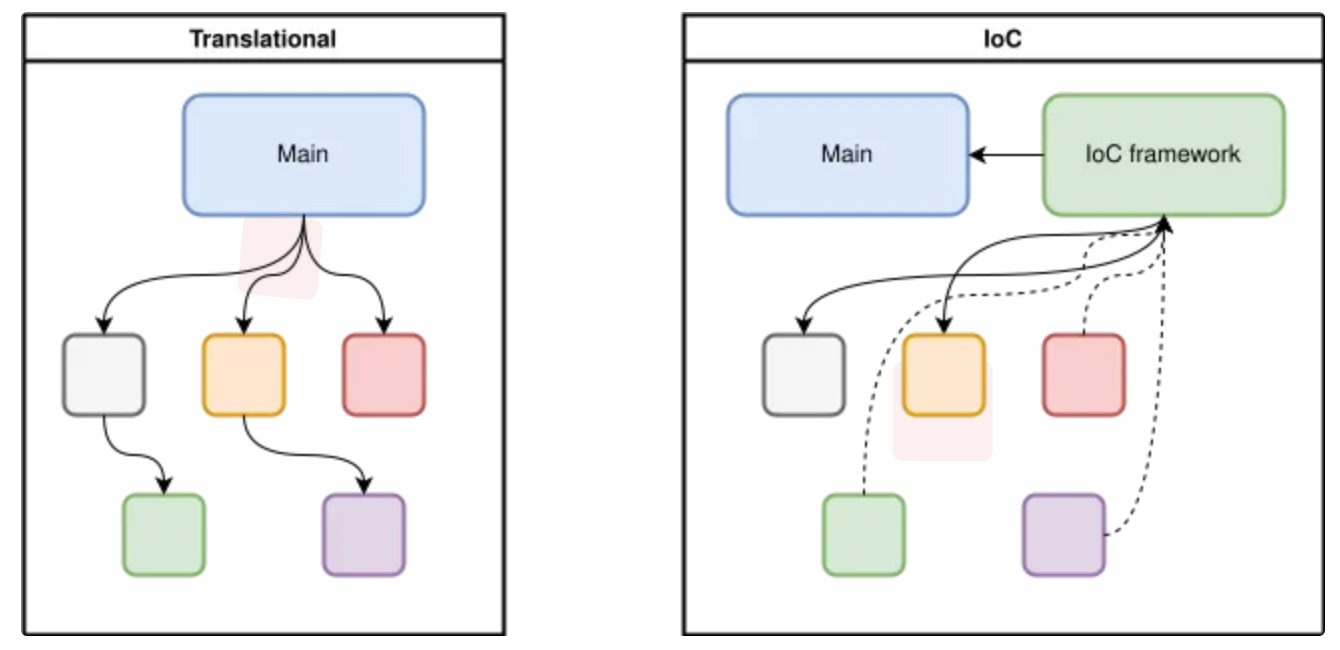
}

}

1. Ưu điểm

* **Giảm sự phụ thuộc chặt chẽ (tight coupling):** DI giúp đối tượng dễ dàng thay thế hoặc mở rộng mà không cần phải thay đổi mã nguồn của các đối tượng khác.
* **Dễ kiểm thử:** Vì phụ thuộc được cung cấp từ bên ngoài, chúng ta có thể thay thế chúng bằng các phiên bản giả (mock) để kiểm thử dễ dàng.
* **Tăng tính mở rộng:** Với DI, khi cần thay đổi cách thực thi của một phụ thuộc, chúng ta chỉ cần thay thế bằng một phụ thuộc khác mà không cần thay đổi đối tượng sử dụng phụ thuộc đó.
* **Dễ bảo trì:** Việc thay đổi phụ thuộc hoặc thêm phụ thuộc mới vào đối tượng được thực hiện dễ dàng mà không làm thay đổi nhiều phần khác của hệ thống.

1. Ví dụ cụ thể như ở loose coupling kia thôi nên e ko viết lại
2. loading ♾️
3. Inversion of control (IoC)



1. Inversion of Control (IoC) là …

* **IoC (Inversion of Control)** là một nguyên tắc mà trong đó quyền điều khiển tạo lập và quản lý các đối tượng được "đảo ngược" từ các đối tượng tự quản lý sang một container hoặc framework. Thay vì các đối tượng tự tạo các phụ thuộc, một **IoC container** sẽ chịu trách nhiệm khởi tạo và quản lý vòng đời của các đối tượng cũng như cung cấp các phụ thuộc khi cần.
* Trong Spring, **IoC container** (thường là ApplicationContext hoặc BeanFactory) sẽ chịu trách nhiệm:
* Khởi tạo các đối tượng (bean).
* Quản lý vòng đời của các đối tượng.
* Tiêm các phụ thuộc vào các đối tượng khi cần.

1. Tại sao cần Inversion of Control

* Trong một ứng dụng lớn, nếu các đối tượng tự quản lý việc khởi tạo các phụ thuộc, điều này sẽ làm tăng sự phụ thuộc giữa các lớp và làm cho mã trở nên khó bảo trì và kiểm thử. Khi một hệ thống có quá nhiều phụ thuộc, việc thay đổi hoặc mở rộng chức năng trở nên khó khăn vì sự thay đổi ở một đối tượng có thể ảnh hưởng đến rất nhiều đối tượng khác.

1. IoC Container

* **IoC container** là thành phần cốt lõi chịu trách nhiệm quản lý các bean trong ứng dụng. Các bean là những đối tượng được quản lý bởi Spring container và có thể được tiêm vào các đối tượng khác thông qua **Dependency Injection (DI)**.
* **IoC container** trong Spring Boot có hai dạng chính:
* **BeanFactory**: IoC container cơ bản nhất, chỉ cung cấp các tính năng quản lý vòng đời bean cơ bản.
* **ApplicationContext**: Mở rộng từ BeanFactory, cung cấp nhiều tính năng hơn như event propagation, declarative mechanism, và tích hợp với Spring AOP.
* loading ♾️
* Spring Boot sử dụng ApplicationContext để quản lý các bean trong ứng dụng
* Các annotation liên quan đến IoC và bean trong Spring Boot
* @Component: Đánh dấu một lớp là bean và được quản lý bởi Spring IoC container.
* @Service: Giống @Component, nhưng chỉ định rằng lớp này là một **service** trong ứng dụng.
* @Repository: Đánh dấu một lớp là bean, thường sử dụng cho các lớp DAO và tương tác với cơ sở dữ liệu.
* @Controller: Đánh dấu một lớp là controller trong ứng dụng MVC.
* @RestController: Tương tự như @Controller, nhưng dùng cho RESTful web service.
* @Autowired: Tiêm phụ thuộc tự động bởi Spring IoC container.
* Quản lý vòng đời của Bean
* **Bean Creation**: Khi ứng dụng khởi chạy, IoC container sẽ tạo các bean.
* **Dependency Injection**: Sau khi tạo bean, các phụ thuộc của nó sẽ được tiêm vào.
* **Initialization**: Các phương thức khởi tạo (nếu có) sẽ được gọi sau khi bean đã được tạo.
* **Destruction**: Khi ứng dụng tắt, Spring IoC container sẽ phá hủy các bean và giải phóng tài nguyên.

Có thể kiểm soát quá trình khởi tạo và hủy bean bằng cách sử dụng các phương thức callback:

* @PostConstruct: Được gọi sau khi bean được tạo và các phụ thuộc đã được tiêm.
* @PreDestroy: Được gọi trước khi bean bị hủy.
* Có thể cấu hình các bean theo cách thủ công trong Spring Boot bằng cách sử dụng annotation @Bean. Đây là cách cấu hình thủ công các bean trong Spring IoC container.

​​import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public MessageService messageService() {

return new EmailMessageService();

}

}

* loading ♾️

1. Ex cho beanFactory và applicationContext
2. beanFactory

* HelloWorld

public class HelloWorld {

private String message;

public void setMessage(String message) {

this.message = message;

}

public void getMessage() {

System.out.println("Print : " + message);

}

}

* beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "stackjava.com.springioc.beanfactory.HelloWorld" >

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

* dùng DefaultListableBeanFactory, một implement của BeanFactory. các version cũ hay sử dụng XmlBeanFactory nhưng nó bị đánh dấu [@Deprecated](https://viblo.asia/u/Deprecated) ở các version mới.

// tạo factory

DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

// đọc thông tin file cấu hình và gán vào factory

XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(new ClassPathResource("beans.xml"));

//tạo đối tượng từ factory

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();

* loading ♾️

1. appliactionContext

* DataResource

public class DataResource {

private String driverClassName;

private String url;

private String username;

private String password;

public String getDriverClassName() {

return driverClassName;

}

public void setDriverClassName(String driverClassName) {

this.driverClassName = driverClassName;

}

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public void printConnection() {

System.out.println("url: " + this.url + "\n" + "username/password: " + this.username + "/" + this.password);

}

}

* bean.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="dataResource" class="stackjava.com.springioc.applicationcontext.DataResource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost/database\_name" />

<property name="username" value="root" />

<property name="password" value="admin1234" />

</bean>

</beans>

* Tạo một đối tượng ApplicationContext

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

DataResource obj = (DataResource) context.getBean("dataResource");

obj.printConnection();

}

1. loading ♾️
2. loangding :v
3. loading ♾️