**BẢNG ĐÁNH GIÁ ĐIỂM THUYẾT TRÌNH**

**MÔN LẬP TRÌNH JAVA NÂNG CAO**

**LỚP CTK44-PM**

|  |  |
| --- | --- |
| **Thông tin đề tài thuyết trình** | **Thông tin sinh viên thực hiện đánh giá** |
| Tên đề tài: Schedule Task, Interceptor và gửi mail | MSSV của sinh viên đánh giá: 2012353 |
| Điểm tổng kết: Ngày trình bày: 12/05/2023 | Họ tên sinh viên & ký xác nhận: Đoàn Cao Nhật Hạ |
| ***Thông tin sinh viên thuyết trình*** |
| MSSV: 2015595  Họ tên: Phạm Ngọc Hưng |

1. **ĐÁNH GIÁ THUYẾT TRÌNH**

**Thang điểm: Kém 0-5; Trung bình: 5-7; Khá 7-8; Tốt 8-9; Xuất sắc: 9-10**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Bảng chấm điểm nhóm thuyết trình** | | | | | | |
| Các chủ đề chính có được xác định? (A) | Slide thuyết trình có thiết kế tốt (B) | Giao tiếp với người nghe bằng ánh mắt (C) | Giao tiếp với người nghe bằng giọng nói (D) | Khả năng trình bày nội dung (E) | Thảo luận & Hỏi đáp (F) | Điểm trung bình = (A+B+C+D+E+F)/6 |
| **9,5** | **9,5** | **9** | **9** | **9** | **9** | **9,2** |

1. **BÀI TẬP NẮM KIẾN THỨC**

Dựa vào nội dung vừa trình bày, trả lời các câu hỏi sau:

1. Trình bày tóm tắt nội dung anh/ chị đã nắm được về chủ đề vừa trình bày?

**Schedule Task**

Đôi khi viết chương trình chúng ta sẽ gặp những tình hướng thực hiện các chức năng chạy theo 1 lịch cố định: ví dụ cứ 5 phút gửi request 1 lần, cứ vào 23h đêm hàng ngày thì thực hiện chạy chức năng backup data, cứ 7h sáng chủ nhật hàng tuần thì tính toán và gửi báo cáo…

Sử dụng TimerTask của Java, các thư viện như Quartz… Trong spring, tích hợp sẵn cho chúng ta.

*Cấu hình Scheduling Tasks trong Spring*

Bật chức năng schedule trong Spring:

Với trường hợp cấu hình bằng annotation hay Spring Boot ta sử dụng annotation @EnableScheduling

@Configuration

@EnableScheduling

public class SpringConfig {

...

}

@SpringBootApplication

@EnableScheduling

public class Application {

...

}

Với trường hợp cấu hình bằng xml ta sử dụng thẻ sau:

<task:annotation-driven>

*Tạo task/job chạy theo lịch với annotation* ***@Schedule***

Tạo schedule task với **fixedDelay**

@Scheduled(fixedDelay = 1000)

public void scheduleFixedDelayTask() throws InterruptedException {

System.out.println("Task1 - " + new Date());

}

* Cứ sau khoảng thời gian fixedDelay thì nó lại chạy một lần, ví dụ ở trên thì cứ sau 1000ms (1 giây) thì nó lại chạy method scheduleFixedDelayTask một lần.
* Với fixedDelay thì chỉ khi nào task trước đó thực hiện xong thì nó mới chạy tiếp task đó lại lần nữa. Ví dụ sau 1 giây mà method scheduleFixedDelayTask chưa chạy xong thì nó sẽ chờ cho tới khi nào xong mới chạy lại lần tiếp theo

Tạo schedule task với **fixedRate**

@Scheduled(fixedRate = 1000)

public void scheduleFixedRateTask() throws InterruptedException {

System.out.println("Task2 - " + new Date());

}

* fixedRate thì giống với fixedDelay, tuy nhiên cứ sau khoảng thời gian fixedRate thì nó chạy tiếp 1 lần nữa mà không cần quan tâm lần chạy trước đã hoàn thành chưa.
* Ví dụ sau 1s mà method scheduleFixedRateTask chưa thực hiện xong thì nó vẫn chạy lần tiếp theo.

Tạo schedule với **cron**

Với cron ta sẽ sử dụng cron expression để định nghĩa lịch chạy.

Bằng cron ta có thể định nghĩa thời gian chạy theo giờ, phút, giây, ngày tháng năm, trong khoảng thời gian nào… do đó việc đặt lịch linh hoạt hơn so với fixedDelay và fixedRate rất nhiều

Ví dụ: từ giây thứ 5 đến giây thứ 10 trong khoảng thời gian 12h-14h các ngày từ thứ 2 đến thứ 6, cứ 1 giây lặp lại một lần

@Scheduled(cron = "5-10/1 \* 12-14 \* \* MON-FRI")

public void scheduleTaskUsingCronExpression() throws InterruptedException {

System.out.println("Task3 - " + new Date());

}

Ngoài việc sử dụng annotation @Scheduled bạn cũng có thể cấu hình trong file xml như sau:

<beans>

<bean name="beanA" class="your\_class"/>

<bean name="beanB" class="your\_class"/>

<bean name="beanC" class="your\_class"/>

</beans>

<!-- Configure parameters -->

<task:scheduled-tasks scheduler="myScheduler">

<task:scheduled ref="beanA" method="scheduleFixedDelayTask"

fixed-delay="1000" initial-delay="1000" />

<task:scheduled ref="beanB" method="scheduleFixedRateTask"

fixed-rate="1000" />

<task:scheduled ref="beanC" method="scheduleTaskUsingCronExpression"

cron="\*/5 \* \* \* \* MON-FRI" />

</task:scheduled-tasks>

*TaskScheduler, thread pool cho schedule task*

Mặc định thread pool cho schedule task có giá trị là 1. Tức là hệ thống chỉ tạo ra duy nhất 1 thread để chạy các schedule task. Do đó bạn sẽ gặp trường hợp đến thời gian chỉ định mà task vẫn không được thực hiện vì có 1 task trước đó chưa hoàn thành kể cả với fixedRate, fixedDelay hay cron

Ví dụ method scheduleFixedRateTask ở trên đang để là 1s chạy 1 lần, nhưng method đó mất tới 5s mới hoàn thành thì phải 5s sau method đó mới được thực hiện một lần nữa.

Để giải quyết vấn đề này ta tăng pool size lên để mỗi task chạy với một thread độc lập:

Tăng pool size cho TaskScheduler bằng cách tạo bean:

@Bean

public TaskScheduler taskScheduler() {

final ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();

scheduler.setPoolSize(10);

return scheduler;

}

Hoặc cấu hình trong file .xml

<task:scheduler id="myScheduler" pool-size="10" />

**Interceptor**

Trong **Spring**, khi một request được gửi đến controller, trước khi request được xử lý bởi **Controller**, nó phải vượt qua các **Interceptor** (0 hoặc nhiều).

**Spring Interceptor** là một khái niệm khá giống với **Servlet Filter**.

**Spring Interceptor** chỉ áp dụng đối với các request đang được gửi đến một **Controller**.

A picture containing text, screenshot, font, flash memory

Description automatically generated

Bạn có thể sử dụng **Interceptor** trong Spring Boot để làm một số việc như:

* Trước khi gửi request tới Controller.
* Trước khi gửi response tới Client.
* Ghi lại Log.

Lớp **Interceptor** của bạn cần phải thực hiện interface **org.springframework.web.servlet.HandlerInterceptor** hoặc mở rộng từ lớp **org.springframework.web.servlet.handler.HandlerInterceptorAdapter**.

Với ví dụ này, bạn có thể sử dụng 1 interceptor để thêm request header trước khi gửi request tới Controller và thêm response header trước khi gửi tới Client.

Để làm việc với Interceptor, bạn cần tạo class **@Component** và được implement từ interface **HandlerInterceptor**.

Sau đây là 3 method bạn nên biết khi làm việc với Interceptor:

A diagram of a login

Description automatically generated with low confidence

* **preHandle()** method: method này sử dụng để thực hiện các operations trước khi gửi request tới Controller. Method này trả về true rồi trả response cho Client.
* **postHandle()** method: method này sử dụng để thực hiện các operations trước khi gửi request tới Client.
* **afterCompletion()** method: method này được sử dụng để thực hiện các operations sau khi hoàn thành việc gửi request và response.

Quan sát đoạn code dưới đây để hiểu rõ hơn.

@Component

**public** **class** **ProductServiceInterceptor** **implements** **HandlerInterceptor** {

@Override

**public** boolean preHandle(

HttpServletRequest request, HttpServletResponse response, Object handler) throws **Exception** {

**return** **true**;

}

@Override

**public** void postHandle(

HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws **Exception** {}

@Override

**public** void afterCompletion(HttpServletRequest request, HttpServletResponse response,

Object handler, **Exception** **exception**) throws **Exception** {}

}

Bạn sẽ phải đăng ký Interceptor này với **InterceptorRegistry** bằng cách sử dụng **WebMvcConfigurerAdapter** như hình dưới đây.

@Component

public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter {

@Autowired

ProductServiceInterceptor productServiceInterceptor;

@Override

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(productServiceInterceptor);

}

}

Trong ví dụ dưới đây, chúng ta sẽ GET ra API của product.

Class ProductServiceInterceptor.java.

package com.tutorialspoint.demo.interceptor;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

@Component

public class ProductServiceInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle

(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

System.out.println("Pre Handle method is Calling");

return true;

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response,

Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("Post Handle method is Calling");

}

@Override

public void afterCompletion

(HttpServletRequest request, HttpServletResponse response, Object

handler, Exception exception) throws Exception {

System.out.println("Request and Response is completed");

}

}

File class Application Configuration để đăng ký Interceptor vào Interceptor Register – ProductServiceInterceptorAppConfig.java.

package com.tutorialspoint.demo.interceptor;

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

import org.springframework.stereotype.Component;

import org.springframework.web.servlet.config.annotation.InterceptorRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Component

public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter {

@Autowired

ProductServiceInterceptor productServiceInterceptor;

@Override

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(productServiceInterceptor);

}

}

File class ProductServiceController.java.

package com.tutorialspoint.demo.controller;

import java.util.HashMap;

import java.util.Map;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

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

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

import com.tutorialspoint.demo.exception.ProductNotfoundException;

import com.tutorialspoint.demo.model.Product;

@RestController

public class ProductServiceController {

private static Map<String, Product> productRepo = new HashMap<>();

static {

Product honey = new Product();

honey.setId("1");

honey.setName("Honey");

productRepo.put(honey.getId(), honey);

Product almond = new Product();

almond.setId("2");

almond.setName("Almond");

productRepo.put(almond.getId(), almond);

}

@RequestMapping(value = "/products")

public ResponseEntity<Object> getProduct() {

return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);

}

}

Model Product.

package com.tutorialspoint.demo.model;

public class Product {

private String id;

private String name;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Hàm main Spring Boot.

package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

Maven build – **pom.xml**.

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

<project xmlns = "http://maven.apache.org/POM/4.0.0" xmlns:xsi = "

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

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.tutorialspoint</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<parent>

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

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

<version>1.5.8.RELEASE</version>

<relativePath/>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Gradle Build – **build.gradle**.

buildscript {

ext {

springBootVersion = '1.5.8.RELEASE'

}

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")

}

}

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

compile('org.springframework.boot:spring-boot-starter-web')

testCompile('org.springframework.boot:spring-boot-starter-test')

}

Bạn có thể tạo file JAR và run Spring Boot bằng cách sử dụng lệnh của Maven hoặc Gradle.

Đối với Maven:

mvn clean install

Sau khi “BUILD SUCCESS”, bạn có thể tìm thấy file JAR trong thư mục đích.

Còn đối với Gradle:

Sau khi “BUILD SUCCESS”, bạn có thể tìm thấy file JAR trong thư mục build/libs.

Run file JAR:

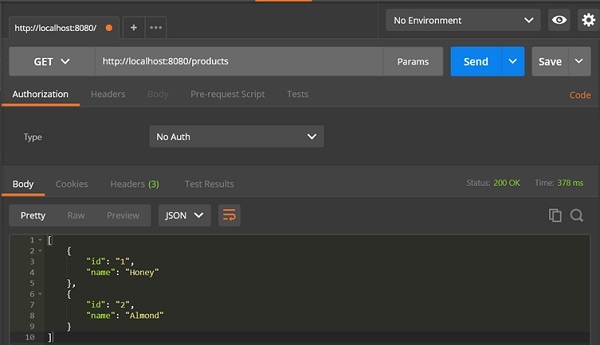
java –jar <JARFILE>

Thao tác này sẽ khởi động cổng ứng dụng Tomcat 8080 như hình dưới.

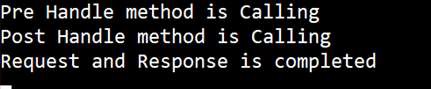
Started Application on Tomcat Port 8080

Started Application on Tomcat Port 8080

Paste link **http://localhost:8080/products** vào POSTMAN, chọn GET rồi gửi request.



Trong Console window, bạn có thể thêm các câu lệnh System.out.println trong Interceptor để được hiển thị như hình dưới nhé.



**Spring Email**

*Spring Email là gì?*

Giới thiệu về Spring Email

**Khái niệm**: Spring Email là một thư viện trong Spring Framework, cung cấp cho các nhà phát triển các lớp và công cụ hỗ trợ để tạo và gửi email trong ứng dụng Spring Boot Java. Thư viện này sử dụng các giao thức SMTP và MIME để tạo và gửi các email, cho phép chúng ta đính kèm các tệp đính kèm, tùy chỉnh nội dung email và gửi email đến nhiều người nhận cùng lúc.

**Tính năng**: Một số tính năng cơ bản của Spring Email.

* Xây dựng nội dung email: Spring Email cung cấp cho chúng ta lớp MimeMessageHelper để dễ dàng tạo nội dung email, cho phép đính kèm tệp tin, chèn hình ảnh, định dạng nội dung email theo HTML hoặc văn bản thuần túy.
* Gửi email đến nhiều người nhận: Chúng ta có thể gửi email đến nhiều người nhận cùng lúc thông qua việc chỉ định danh sách địa chỉ email của các người nhận.
* Tính năng đa luồng: Spring Email hỗ trợ tính năng đa luồng để xử lý các nhiệm vụ gửi email một cách hiệu quả và không làm chậm hoạt động của ứng dụng.
* Tính năng tùy chỉnh cao: Spring Email cho phép chúng ta tùy chỉnh các thông tin về SMTP server, cấu hình TLS/SSL, xác thực, thời gian chờ kết nối, các thông tin xác thực tài khoản email, v.v. để phù hợp với yêu cầu của ứng dụng.

**Ứng dụng**: Spring Email được sử dụng rộng rãi trong các ứng dụng web để gửi các email thông báo đến người dùng, như thông báo đặt hàng, thông báo kích hoạt tài khoản, v.v. Ngoài ra, nó cũng được sử dụng trong các ứng dụng quản lý email như các ứng dụng webmail hoặc các công cụ quản lý email khác. Với tính năng tùy chỉnh cao, Spring Email cũng có thể được sử dụng để gửi email tùy chỉnh trong các ứng dụng phát triển khác như ứng dụng máy chủ hoặc ứng dụng máy tính cá nhân.

*Tầm quan trọng của Spring Email trong ứng dụng web?*

Với tầm quan trọng của email trong giao tiếp trực tuyến hiện nay, Spring Email có tầm quan trọng rất lớn trong ứng dụng web. Dưới đây là một số tầm quan trọng của Spring Email trong ứng dụng web:

**Gửi thông báo cho người dùng**: Trong ứng dụng web, chúng ta thường cần gửi email thông báo cho người dùng để cập nhật về các sự kiện quan trọng, ví dụ như đăng ký tài khoản, kích hoạt tài khoản, thông báo đặt hàng, v.v. Spring Email cung cấp các công cụ để tạo và gửi các email thông báo này một cách dễ dàng và nhanh chóng.

**Gửi email xác nhận**: Khi người dùng đăng ký tài khoản hoặc thực hiện một hành động quan trọng, chúng ta thường cần gửi email xác nhận để đảm bảo tính xác thực và bảo mật cho ứng dụng. Spring Email cung cấp các công cụ để tạo và gửi email xác nhận một cách dễ dàng và đáng tin cậy.

**Gửi email marketing**: Trong ứng dụng web, chúng ta có thể sử dụng email để quảng cáo sản phẩm hoặc dịch vụ đến khách hàng. Spring Email cung cấp các công cụ để tạo và gửi các email marketing này một cách dễ dàng và tiện lợi.

**Quản lý email**: Nếu ứng dụng web cần tích hợp chức năng quản lý email như các ứng dụng webmail hoặc các công cụ quản lý email khác, Spring Email là một lựa chọn tuyệt vời để hỗ trợ việc này. Chúng ta có thể sử dụng Spring Email để tạo và gửi email, cũng như quản lý các email đã gửi hoặc nhận.

* Với các tính năng tiên tiến và tùy chỉnh cao, Spring Email giúp các nhà phát triển ứng dụng web xây dựng và triển khai các tính năng email một cách dễ dàng và hiệu quả.

*Cài đặt và cấu hình môi trường để sử dụng Spring Email*

**Bước 1**: Tạo project.

Đầu tiên các bạn cần tạo mới một dự án mới, bạn có thể tham khảo các bước tạo project trên phần mềm Eclipse qua các bài trước.

Hoặc bạn có thể tạo project trên trang chủ Spring Boot Initializr hoặc sử dụng các IDE như Eclipse, IntelliJ IDEA, hoặc NetBeans để tạo project.

**Bước 2**: Thêm dependency Spring Email

Thêm dependency của Spring Email vào file pom.xml để có thể sử dụng được Spring Email:

<dependency>

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

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

</dependency>

**Bước 2**: Cấu hình file

Cấu hình thông tin kết nối SMTP server trong file **application.properties** hoặc **application.yml**.

Ví dụ:

spring.mail.host=smtp.gmail.com

spring.mail.port=587

spring.mail.username=example@gmail.com

spring.mail.password=examplepassword

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

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

Lưu ý: Chúng ta có thể thay đổi thông tin kết nối SMTP server tùy theo nhu cầu sử dụng.

*Sử dụng Spring Email*

Tạo một đối tượng EmailSender và cấu hình nó trong Spring Boot

Để cấu hình EmailSender trong Spring Boot thì chúng ta cần add dependency và cấu hình thông tin kết nối như phần trên mình vừa hướng dẫn.

**Bước 1**: Tạo đối tượng EmailSender

Để tạo đối tượng EmailSender, bạn có thể sử dụng @Autowired để inject dependency của EmailSender vào trong class của bạn.

Ví dụ:

@Service

public class MyEmailService {

@Autowired

private JavaMailSender javaMailSender;

// ...

}

**Bước 2**: Sử dụng đối tượng EmailSender để gửi email

Sau khi tạo đối tượng EmailSender, bạn có thể sử dụng các phương thức của nó để gửi email.

Ví dụ:

@Service

public class MyEmailService {

@Autowired

private JavaMailSender javaMailSender;

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

SimpleMailMessage message = new SimpleMailMessage();

message.setTo(to);

message.setSubject(subject);

message.setText(text);

javaMailSender.send(message);

}

}

Trong ví dụ trên, chúng ta đã tạo một đối tượng SimpleMailMessage để lưu trữ các thông tin cần thiết để gửi email, sau đó sử dụng phương thức javaMailSender.send() để gửi email.

Tạo một đối tượng MimeMessagePreparator để thiết lập nội dung email

Để tạo một đối tượng **MimeMessagePreparator** trong Spring Boot để thiết lập nội dung email, bạn có thể sử dụng đối tượng MimeMessageHelper để tạo email có định dạng **MIME**. MimeMessageHelper cung cấp các phương thức để thiết lập các thuộc tính của email như địa chỉ email người gửi, địa chỉ email người nhận, tiêu đề email và nội dung email.

Ví dụ, để tạo một đối tượng MimeMessagePreparator với nội dung email là một chuỗi HTML, bạn có thể sử dụng đoạn mã sau:

MimeMessagePreparator messagePreparator = mimeMessage -> {

MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage, true, "UTF-8");

messageHelper.setFrom("sender@example.com");

messageHelper.setTo("recipient@example.com");

messageHelper.setSubject("Test email");

String htmlContent = "<html><body><h1>This is a test email</h1></body></html>";

messageHelper.setText(htmlContent, true);

};

Lưu ý rằng trong ví dụ trên, chúng ta đã sử dụng charset là UTF-8 để đảm bảo rằng email có thể hiển thị đúng các ký tự Unicode.

Sau khi tạo đối tượng MimeMessagePreparator, bạn có thể sử dụng nó để gửi email bằng cách sử dụng phương thức send() của JavaMailSender:

javaMailSender.send(messagePreparator);

Khi gửi email, JavaMailSender sẽ sử dụng thông tin cấu hình được cung cấp trong file **application.properties** hoặc **application.yml** để thiết lập kết nối SMTP và gửi email.

*Ví dụ gửi Email đơn giản*

Tạo một ứng dụng Spring Boot đơn giản để gửi email

**Bước 1**: Các bước tạo project và cấu hình dự án các bạn có thể tham khảo ở các bước trên.

**Bước 2**: Tiếp đó là bước tạo JavaMailSender

Tạo một đối tượng JavaMailSender trong ứng dụng của bạn bằng cách sử dụng @Autowired và @Bean annotation:

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.mail.javamail.JavaMailSender;

import org.springframework.mail.javamail.JavaMailSenderImpl;

@Configuration

public class MailConfig {

    @Autowired

    private MailProperties mailProperties;

    @Bean

    public JavaMailSender javaMailSender() {

        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();

        mailSender.setHost(mailProperties.getHost());

        mailSender.setPort(mailProperties.getPort());

        mailSender.setUsername(mailProperties.getUsername());

        mailSender.setPassword(mailProperties.getPassword());

        mailSender.setJavaMailProperties(mailProperties.getProperties());

        return mailSender;

    }

}

Lưu ý rằng trong ví dụ trên, chúng ta đã sử dụng **@Autowired** để inject đối tượng **MailProperties**, đây là một lớp để lưu trữ các thông tin cấu hình của email được cấu hình trong file **application.properties**.

**Bước 3**: Tạo một đối tượng MimeMessagePreparator để thiết lập nội dung email

Tạo một đối tượng MimeMessagePreparator để thiết lập nội dung email của bạn.

Ví dụ:

import org.springframework.mail.javamail.MimeMessageHelper;

import org.springframework.mail.javamail.MimeMessagePreparator;

import org.thymeleaf.TemplateEngine;

import org.thymeleaf.context.Context;

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

import org.springframework

Tiếp tục phần trước, bạn có thể sử dụng Thymeleaf để tạo email với định dạng HTML. Thymeleaf là một công cụ mạnh mẽ cho phép bạn tạo các mẫu HTML động. Để sử dụng Thymeleaf trong ứng dụng của bạn, bạn cần thêm dependency **spring-boot-starter-thymeleaf** vào file **pom.xml** hoặc **build.gradle**.

**Bước 4**: Tạo một template email bằng Thymeleaf

Bạn có thể tạo một template email bằng Thymeleaf, tại đây bạn có thể sử dụng các biểu thức Thymeleaf để tạo nội dung email động.

Ví dụ:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Mail Template</title>

</head>

<body>

<p th:text="${message}"></p>

</body>

</html>

Trong ví dụ trên, chúng ta sử dụng biểu thức Thymeleaf ${message} để đưa dữ liệu động vào template email

**Bước 5**: Tạo đối tượng MimeMessagePreparator

Sau khi tạo template email, bạn có thể tạo đối tượng MimeMessagePreparator để thiết lập nội dung email.

Ví dụ:

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

import org.springframework.mail.javamail.MimeMessageHelper;

import org.springframework.mail.javamail.MimeMessagePreparator;

import org.springframework.stereotype.Component;

import org.thymeleaf.TemplateEngine;

import org.thymeleaf.context.Context;

import javax.mail.internet.MimeMessage;

@Component

public class EmailService {

    @Autowired

    private JavaMailSender javaMailSender;

    @Autowired

    private TemplateEngine templateEngine;

    public void sendMail(String to, String subject, String message) {

        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws Exception {

                MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage, true, "UTF-8");

                messageHelper.setFrom("noreply@example.com");

                messageHelper.setTo(to);

                messageHelper.setSubject(subject);

                Context context = new Context();

                context.setVariable("message", message);

                String content = templateEngine.process("email-template", context);

                messageHelper.setText(content, true);

            }

        };

        javaMailSender.send(preparator);

    }

}

Trong ví dụ trên, chúng ta sử dụng đối tượng **TemplateEngine** để render nội dung email từ template Thymeleaf. Sau đó, chúng ta sử dụng đối tượng **MimeMessageHelper** để thiết lập thông tin email, bao gồm địa chỉ email người nhận, chủ đề email và nội dung email. Cuối cùng, chúng ta sử dụng đối tượng **JavaMailSender** để gửi email.

**Bước 7**: Gửi Email

Cuối cùng, bạn có thể gửi email bằng cách gọi phương thức sendMail() của đối tượng EmailService.

Ví dụ:

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.annotation.Bean;

import org.springframework.mail.javamail.JavaMailSender;

import org.springframework.mail.javamail.JavaMailSenderImpl;

@SpringBootApplication

public class EmailApplication {

    public static void main(String[] args) {

        ConfigurableApplicationContext context = SpringApplication.run(EmailApplication.class, args);

        EmailService emailService = context.getBean(EmailService.class);

        emailService.sendMail("to@example.com", "Test Email", "This is a test email.");

    }

    @Bean

    public JavaMailSender javaMailSender() {

        JavaMailSenderImpl javaMailSender = new JavaMailSenderImpl();

        javaMailSender.setHost("smtp.gmail.com");

        javaMailSender.setPort(587);

        javaMailSender.setUsername("your-email@gmail.com");

        javaMailSender.setPassword("your-password");

        Properties properties = javaMailSender.getJavaMailProperties();

        properties.put("mail.transport.protocol", "smtp");

        properties.put("mail.smtp.auth", "true");

        properties.put("mail.smtp.starttls.enable", "true");

        properties.put("mail.debug", "true");

        return javaMailSender;

    }

}

Nếu bạn sử dụng Gmail để gửi email, bạn cần bật chế độ truy cập cho ứng dụng kém an toàn bằng cách vào mục Security của tài khoản Google và bật tùy chọn "Less secure app access".

Thực hiện gửi email thông qua giao diện người dùng hoặc REST API

Để gửi email thông qua giao diện người dùng hoặc **REST API** trong Spring Email trong Spring Boot, chúng ta có thể sử dụng các framework như Thymeleaf để tạo giao diện và Spring MVC hoặc Spring WebFlux để xử lý các yêu cầu REST API.

**Bước 1**: Sử dụng Thymeleaf để tạo giao diện người dùng

Đầu tiên, chúng ta cần tạo một trang HTML sử dụng Thymeleaf để cho phép người dùng nhập thông tin và gửi email.

Dưới đây là một ví dụ đơn giản:

<!DOCTYPE html>

<html xmlns:th="[http://www.thymeleaf.org](http://www.thymeleaf.org/)">

<head>

    <meta charset="UTF-8">

    <title>Send Email</title>

</head>

<body>

    <h1>Send Email</h1>

    <form method="post" th:action="@{/send-email}">

        <label for="to">To:</label>

        <input type="email" id="to" name="to" required><br>

        <label for="subject">Subject:</label>

        <input type="text" id="subject" name="subject" required><br>

        <label for="body">Body:</label>

        <textarea id="body" name="body" rows="5" required></textarea><br>

        <button type="submit">Send</button>

    </form>

</body>

</html>

Ở đây, chúng ta sử dụng Thymeleaf để tạo form và định dạng trang HTML. Đường dẫn đến phương thức xử lý yêu cầu gửi email sử dụng thuộc tính th:action="@{/send-email}".

**Bước 2**: Tạo một phương thức xử lý yêu cầu

Sau đó, chúng ta cần tạo một phương thức xử lý yêu cầu POST để gửi email. Ở đây, chúng ta sử dụng đối tượng EmailService đã được cấu hình trong phần trước để gửi email:

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

import org.springframework.stereotype.Controller;

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

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

@Controller

public class EmailController {

    @Autowired

    private EmailService emailService;

    @PostMapping("/send-email")

    public String sendEmail(@RequestParam String to, @RequestParam String subject, @RequestParam String body) {

        emailService.sendMail(to, subject, body);

        return "redirect:/";

    }

}

Ở đây, chúng ta sử dụng **@PostMapping** để xử lý yêu cầu POST từ form và sử dụng đối tượng EmailService để gửi email. Sau khi gửi email thành công, chúng ta chuyển hướng trở lại trang chủ.

**Bước 3**: Sử dụng Spring MVC hoặc Spring WebFlux để tạo REST API

Đối với việc gửi email thông qua REST API, chúng ta có thể sử dụng Spring MVC hoặc Spring WebFlux để xử lý các yêu cầu HTTP.

Dưới đây là một ví dụ sử dụng Spring MVC:

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

@RestController

public class EmailController {

    @Autowired

    private EmailService emailService;

    @PostMapping("/send-email")

    public ResponseEntity<String> sendEmail(@RequestBody EmailRequest emailRequest) {

        emailService.sendMail(emailRequest.getTo(), emailRequest.getSubject(), emailRequest.getBody());

        return new ResponseEntity<>("Email sent successfully", HttpStatus.OK);

    }

}

**Bước 4**: Với Spring WebFlux, chúng ta có thể sử dụng đối tượng WebClient để gửi yêu cầu HTTP đến một REST API khác để gửi email.

Dưới đây là một ví dụ:

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

import org.springframework.http.MediaType;

import org.springframework.stereotype.Controller;

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

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

import org.springframework.web.reactive.function.client.WebClient;

import reactor.core.publisher.Mono;

@Controller

public class EmailController {

    @Autowired

    private WebClient webClient;

    @PostMapping("/send-email")

    public Mono<String> sendEmail(@RequestBody EmailRequest emailRequest) {

        return webClient.post()

                .uri("/send-email")

                .contentType(MediaType.APPLICATION\_JSON)

                .body(Mono.just(emailRequest), EmailRequest.class)

                .retrieve()

                .bodyToMono(String.class);

    }

}

Ở đây, chúng ta sử dụng WebClient để gửi yêu cầu POST đến một REST API khác để gửi email. Chúng ta cũng sử dụng Mono để xử lý kết quả trả về từ REST API.

**Bước 5**: Ngoài ra, bạn cũng có thể tích hợp email vào các ứng dụng Spring Boot khác như các ứng dụng hàng đợi bằng cách sử dụng Spring AMQP hoặc các ứng dụng lên lịch bằng cách sử dụng Spring Scheduling.

Ví dụ, bạn có thể tạo một Job để gửi email mỗi ngày lúc 9 giờ sáng như sau:

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

import org.springframework.scheduling.annotation.Scheduled;

import org.springframework.stereotype.Component;

@Component

public class EmailJob {

    @Autowired

    private EmailService emailService;

    @Scheduled(cron = "0 0 9 \* \* \*")

    public void sendDailyEmail() {

        String to = "example@example.com";

        String subject = "Daily Report";

        String body = "Hello,\n\nHere is your daily report.\n\nBest regards,\nExample team";

        emailService.sendMail(to, subject, body);

    }

}

Ở đây, chúng ta sử dụng **@Scheduled** để lên lịch Job để gửi email mỗi ngày lúc 9 giờ sáng. Đối tượng EmailService được sử dụng để gửi email.

1. Anh/ chị hãy đưa ra các nhận xét và góp ý cho sinh viên vừa trình bày?

Giải thích dễ hiểu hơn chút