Sure! Let's create a Spring Boot 3 application for policy management. This application will include CRUD operations to manage policy information. Here's a step-by-step guide to get you started:

**Step 1: Set Up Your Spring Boot Project**

1. **Create a new Spring Boot project** using Spring Initializr:
   * Project: Maven
   * Language: Java
   * Spring Boot: 3.x.x
   * Dependencies: Spring Web, Spring Data JPA, H2 Database (for simplicity)
2. **Download the project** and open it in your favorite IDE (e.g., IntelliJ IDEA, Eclipse).

**Step 2: Define the Policy Entity**

Create a Policy class in the model package to represent the policy information.

package com.example.policy.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Policy {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String policyNumber;

private String policyType;

private String coverageDetails;

private String contactInfo;

// Getters and Setters

}

**Step 3: Create the Policy Repository**

Create a PolicyRepository interface in the repository package to handle database operations.

package com.example.policy.repository;

import com.example.policy.model.Policy;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PolicyRepository extends JpaRepository<Policy, Long> {

}

**Step 4: Create the Policy Service**

Create a PolicyService class in the service package to handle business logic.

package com.example.policy.service;

import com.example.policy.model.Policy;

import com.example.policy.repository.PolicyRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class PolicyService {

@Autowired

private PolicyRepository policyRepository;

public List<Policy> getAllPolicies() {

return policyRepository.findAll();

}

public Optional<Policy> getPolicyById(Long id) {

return policyRepository.findById(id);

}

public Policy savePolicy(Policy policy) {

return policyRepository.save(policy);

}

public void deletePolicy(Long id) {

policyRepository.deleteById(id);

}

}

**Step 5: Create the Policy Controller**

Create a PolicyController class in the controller package to handle HTTP requests.

package com.example.policy.controller;

import com.example.policy.model.Policy;

import com.example.policy.service.PolicyService;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/policies")

public class PolicyController {

@Autowired

private PolicyService policyService;

@GetMapping

public List<Policy> getAllPolicies() {

return policyService.getAllPolicies();

}

@GetMapping("/{id}")

public ResponseEntity<Policy> getPolicyById(@PathVariable Long id) {

Optional<Policy> policy = policyService.getPolicyById(id);

return policy.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

@PostMapping

public Policy createPolicy(@RequestBody Policy policy) {

return policyService.savePolicy(policy);

}

@PutMapping("/{id}")

public ResponseEntity<Policy> updatePolicy(@PathVariable Long id, @RequestBody Policy policyDetails) {

Optional<Policy> policy = policyService.getPolicyById(id);

if (policy.isPresent()) {

Policy updatedPolicy = policy.get();

updatedPolicy.setPolicyNumber(policyDetails.getPolicyNumber());

updatedPolicy.setPolicyType(policyDetails.getPolicyType());

updatedPolicy.setCoverageDetails(policyDetails.getCoverageDetails());

updatedPolicy.setContactInfo(policyDetails.getContactInfo());

return ResponseEntity.ok(policyService.savePolicy(updatedPolicy));

} else {

return ResponseEntity.notFound().build();

}

}

@DeleteMapping("/{id}")

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

policyService.deletePolicy(id);

return ResponseEntity.noContent().build();

}

}

**Step 6: Configure the Application Properties**

Add the following configuration to your application.properties file to set up the H2 database.

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

**Step 7: Run the Application**

Run the Spring Boot application and test the CRUD operations using tools like Postman or cURL.

That's it! You now have a basic Spring Boot 3 application for managing policy information. If you have any questions or need further assistance, feel free to ask!

Sure! Let's write unit and integration tests using JUnit 5, which is compatible with Spring Boot 3.4.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Tests**

Unit tests focus on individual components. Here’s an example for the PolicyService class.

package com.example.policy.service;

import com.example.policy.model.Policy;

import com.example.policy.repository.PolicyRepository;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

class PolicyServiceTest {

@Mock

private PolicyRepository policyRepository;

@InjectMocks

private PolicyService policyService;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testGetPolicyById() {

Policy policy = new Policy();

policy.setId(1L);

when(policyRepository.findById(1L)).thenReturn(Optional.of(policy));

Optional<Policy> foundPolicy = policyService.getPolicyById(1L);

assertEquals(1L, foundPolicy.get().getId());

}

@Test

void testSavePolicy() {

Policy policy = new Policy();

when(policyRepository.save(any(Policy.class))).thenReturn(policy);

Policy savedPolicy = policyService.savePolicy(policy);

verify(policyRepository, times(1)).save(policy);

}

}

**Step 3: Create Integration Tests**

Integration tests ensure that different parts of the application work together. Here’s an example using @SpringBootTest and MockMvc.

package com.example.policy.controller;

import com.example.policy.model.Policy;

import com.example.policy.repository.PolicyRepository;

import org.junit.jupiter.api.Test;

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

import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@SpringBootTest

@AutoConfigureMockMvc

class PolicyControllerIntegrationTest {

@Autowired

private MockMvc mockMvc;

@Autowired

private PolicyRepository policyRepository;

@Test

void testGetAllPolicies() throws Exception {

mockMvc.perform(get("/api/policies"))

.andExpect(status().isOk())

.andExpect(content().contentType(MediaType.APPLICATION\_JSON));

}

@Test

void testCreatePolicy() throws Exception {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

mockMvc.perform(post("/api/policies")

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"policyNumber\":\"12345\",\"policyType\":\"Health\",\"coverageDetails\":\"Full Coverage\",\"contactInfo\":\"contact@example.com\"}"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.policyNumber").value("12345"));

}

}

**Step 4: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your application is working correctly. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [Spring Boot 3 with JUnit 5 - HowToDoInJava](https://howtodoinjava.com/spring-boot/junit-with-spring-boot/)

[2] [Testing in Spring Boot - Baeldung](https://www.baeldung.com/spring-boot-testing)

[3] [Spring Boot Tests for Controller, Service and Dao Layers - HowToDoInJava](https://howtodoinjava.com/spring-boot/spring-boot-test-controller-service-dao/)

It looks like you're using Lombok to generate getters and setters, but they aren't being recognized. Here are some steps to ensure Lombok works correctly in your project:

**Step 1: Add Lombok Dependency**

Ensure you have the Lombok dependency in your pom.xml.

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.24</version>

<scope>provided</scope>

</dependency>

**Step 2: Install Lombok Plugin in Your IDE**

If you're using IntelliJ IDEA:

1. Go to File -> Settings -> Plugins.
2. Search for "Lombok" and install the plugin.
3. Restart IntelliJ IDEA.

**Step 3: Enable Annotation Processing**

In IntelliJ IDEA:

1. Go to File -> Settings -> Build, Execution, Deployment -> Compiler -> Annotation Processors.
2. Check the box for "Enable annotation processing".

**Step 4: Clean and Rebuild Your Project**

Sometimes, the IDE needs to reindex the project to recognize Lombok annotations. Try cleaning and rebuilding your project:

1. In IntelliJ IDEA, go to Build -> Rebuild Project.
2. Alternatively, you can use the Maven tool window to run mvn clean install.

**Step 5: Verify Your Entity Class**

Ensure your Policy entity class is correctly annotated and doesn't have redundant getter and setter methods.

package com.imsapp.ims.entity.policy;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import lombok.Getter;

import lombok.Setter;

@Entity

@Getter

@Setter

public class Policy {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String policyNumber;

private String policyType;

private String coverageDetails;

private String contactInfo;

}

**Step 6: Check for IDE-Specific Issues**

If the issue persists, it might be related to your IDE configuration. Here are some additional steps:

1. **Invalidate Caches/Restart**: In IntelliJ IDEA, go to File -> Invalidate Caches / Restart.
2. **Reimport Project**: Use the Maven tool window to reimport the project by clicking the refresh button.

By following these steps, Lombok should correctly generate the getter and setter methods for your Policy class. If you still encounter issues, please let me know, and we can troubleshoot further!

**References**

[1] [Lombok added but getters and setters not recognized in Intellij IDEA](https://stackoverflow.com/questions/17729384/lombok-added-but-getters-and-setters-not-recognized-in-intellij-idea)

[2] [Lombok is not generating getter and setter - Stack Overflow](https://stackoverflow.com/questions/11803948/lombok-is-not-generating-getter-and-setter)

[3] [Lombok Not Working as Expected in IntelliJ with Maven ... - GitHub](https://github.com/spring-projects/spring-boot/issues/43593)

[4] [[BUG] 'Symbol' not recognized. Getters and Setters. #3804 - GitHub](https://github.com/projectlombok/lombok/issues/3804)

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

public class PolicyRequest {

private String policyNumber;

private String policyType;

private String coverageDetails;

private String contactInfo;

}

@Getter  
@Setter  
public class PolicyResponse {  
 private Long id;  
 private String policyNumber;  
 private String policyType;  
 private String coverageDetails;  
 private String contactInfo;  
}

Using request and response objects can help in structuring your API endpoints more clearly and can be beneficial for validation and transformation purposes. However, for simple CRUD operations, you can directly use the entity class. If you want to use request and response objects, here's how you can do it:

**Step 1: Create Request and Response DTOs**

Create Data Transfer Objects (DTOs) for request and response in the dto package.

package com.imsapp.ims.dto;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

public class PolicyRequest {

private String policyNumber;

private String policyType;

private String coverageDetails;

private String contactInfo;

}

@Getter

@Setter

public class PolicyResponse {

private Long id;

private String policyNumber;

private String policyType;

private String coverageDetails;

private String contactInfo;

}

**Step 2: Update the Controller**

Modify the PolicyController to use the request and response DTOs.

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.PolicyRequest;

import com.imsapp.ims.dto.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.service.PolicyService;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

@RestController

@Requestapi/policies")

public class PolicyController {

@Autowired

private PolicyService policyService;

@GetMapping

public List<PolicyResponse> getAllPolicies() {

return policyService.getAllPolicies().stream().map(this::convertToResponse).collect(Collectors.toList());

}

@GetMapping("/{id}")

public ResponseEntity<PolicyResponse> getPolicyById(@PathVariable Long id) {

Optional<Policy> policy = policyService.getPolicyById(id);

return policy.map(value -> ResponseEntity.ok(convertToResponse(value))).orElseGet(() -> ResponseEntity.notFound().build());

}

@PostMapping

public PolicyResponse createPolicy(@RequestBody PolicyRequest policyRequest) {

Policy policy = convertToEntity(policyRequest);

return convertToResponse(policyService.savePolicy(policy));

}

@PutMapping("/{id}")

public ResponseEntity<PolicyResponse> updatePolicy(@PathVariable Long id, @RequestBody PolicyRequest policyRequest) {

Optional<Policy> policy = policyService.getPolicyById(id);

if (policy.isPresent()) {

Policy updatedPolicy = policy.get();

updatedPolicy.setPolicyNumber(policyRequest.getPolicyNumber());

updatedPolicy.setPolicyType(policyRequest.getPolicyType());

updatedPolicy.setCoverageDetails(policyRequest.getCoverageDetails());

updatedPolicy.setContactInfo(policyRequest.getContactInfo());

return ResponseEntity.ok(convertToResponse(policyService.savePolicy(updatedPolicy)));

} else {

return ResponseEntity.notFound().build();

}

}

@DeleteMapping("/{id}")

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

policyService.deletePolicy(id);

return ResponseEntity.noContent().build();

}

private Policy convertToEntity(PolicyRequest policyRequest) {

Policy policy = new Policy();

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return policy;

}

private PolicyResponse convertToResponse(Policy policy) {

PolicyResponse policyResponse = new PolicyResponse();

policyResponse.setId(policy.getId());

policyResponse.setPolicyNumber(policy.getPolicyNumber());

policyResponse.setPolicyType(policy.getPolicyType());

policyResponse.setCoverageDetails(policy.getCoverageDetails());

policyResponse.setContactInfo(policy.getContactInfo());

return policyResponse;

}

}

**Step 3: Update the Service and Repository**

No changes are needed in the service and repository layers as they will continue to work with the Policy entity.

**Step 4: Test the Application**

Run your application and test the endpoints using tools like Postman or cURL to ensure everything works as expected.

Using request and response DTOs can help in decoupling your API layer from the entity layer, making your application more maintainable and flexible. If you have any further questions or need additional assistance, feel free to ask!

Global Exception handler

Sure! Adding a global exception handler to your Spring Boot application can help manage errors consistently across your application. Here's how you can do it:

**Step 1: Create a Custom Exception Class**

First, create a custom exception class if you need specific exceptions.

package com.imsapp.ims.exception;

public class PolicyNotFoundException extends RuntimeException {

public PolicyNotFoundException(String message) {

super(message);

}

}

**Step 2: Create an Error Response Class**

Create a class to represent the error response.

package com.imsapp.ims.dto;

import lombok.Getter;

import lombok.Setter;

import org.springframework.http.HttpStatus;

import java.time.LocalDateTime;

@Getter

@Setter

public class ErrorResponse {

private LocalDateTime timestamp;

private HttpStatus status;

private String message;

private String path;

public ErrorResponse(HttpStatus status, String message, String path) {

this.timestamp = LocalDateTime.now();

this.status = status;

this.message = message;

this.path = path;

}

}

**Step 3: Create a Global Exception Handler**

Create a global exception handler using @ControllerAdvice.

package com.imsapp.ims.exception;

import com.imsapp.ims.dto.ErrorResponse;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

import org.springframework.web.context.request.WebRequest;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(PolicyNotFoundException.class)

public ResponseEntity<ErrorResponse> handlePolicyNotFoundException(PolicyNotFoundException ex, WebRequest request) {

ErrorResponse errorResponse = new ErrorResponse(HttpStatus.NOT\_FOUND, ex.getMessage(), request.getDescription(false));

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

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex, WebRequest request) {

ErrorResponse errorResponse = new ErrorResponse(HttpStatus.INTERNAL\_SERVER\_ERROR, ex.getMessage(), request.getDescription(false));

return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**Step 4: Update the Controller to Use Custom Exceptions**

Modify your controller to throw custom exceptions where appropriate.

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.PolicyRequest;

import com.imsapp.ims.dto.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.exception.PolicyNotFoundException;

import com.imsapp.ims.service.PolicyService;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/api/policies")

public class PolicyController {

@Autowired

private PolicyService policyService;

@GetMapping

public List<PolicyResponse> getAllPolicies() {

return policyService.getAllPolicies().stream().map(this::convertToResponse).collect(Collectors.toList());

}

@GetMapping("/{id}")

public ResponseEntity<PolicyResponse> getPolicyById(@PathVariable Long id) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

return ResponseEntity.ok(convertToResponse(policy));

}

@PostMapping

public PolicyResponse createPolicy(@RequestBody PolicyRequest policyRequest) {

Policy policy = convertToEntity(policyRequest);

return convertToResponse(policyService.savePolicy(policy));

}

@PutMapping("/{id}")

public ResponseEntity<PolicyResponse> updatePolicy(@PathVariable Long id, @RequestBody PolicyRequest policyRequest) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return ResponseEntity.ok(convertToResponse(policyService.savePolicy(policy)));

}

@DeleteMapping("/{id}")

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

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policyService.deletePolicy(id);

return ResponseEntity.noContent().build();

}

private Policy convertToEntity(PolicyRequest policyRequest) {

Policy policy = new Policy();

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return policy;

}

private PolicyResponse convertToResponse(Policy policy) {

PolicyResponse policyResponse = new PolicyResponse();

policyResponse.setId(policy.getId());

policyResponse.setPolicyNumber(policy.getPolicyNumber());

policyResponse.setPolicyType(policy.getPolicyType());

policyResponse.setCoverageDetails(policy.getCoverageDetails());

policyResponse.setContactInfo(policy.getContactInfo());

return policyResponse;

}

}

**Step 5: Test the Application**

Run your application and test the endpoints using tools like Postman or cURL to ensure the global exception handler works as expected.

By following these steps, you can add a global exception handler to your Spring Boot application, making error handling more consistent and informative for the client. If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Custom Error Message Handling for REST API | Baeldung](https://www.baeldung.com/global-error-handler-in-a-spring-rest-api)

[2] [Global Exception Handler in Spring Boot - DEV Community](https://dev.to/tienbku/global-exception-handler-in-spring-boot-3mbp)

[3] [Best way to add the custom exception for the spring boot code](https://stackoverflow.com/questions/52437023/best-way-to-add-the-custom-exception-for-the-spring-boot-code)

Testing your Spring Boot application using SoapUI involves creating and sending requests to your application's endpoints and verifying the responses. Here's a step-by-step guide to test your policy management application using SoapUI:

**Step 1: Install SoapUI**

1. **Download and install SoapUI** from the official website.

**Step 2: Create a New Project in SoapUI**

1. **Open SoapUI** and create a new project:
   * Click on File -> New SOAP Project.
   * Enter a name for your project.
   * Optionally, you can specify the WSDL URL if you are testing a SOAP web service.

**Step 3: Define REST API Endpoints**

Since your application uses REST endpoints, you need to define them in SoapUI:

1. **Right-click on the project** and select New REST Service.
2. **Enter the base URL** of your Spring Boot application (e.g., http://localhost:8080/api/policies).

**Step 4: Create Test Requests**

1. **Create a new request** for each endpoint you want to test:
   * **GET Request**: To retrieve all policies.
     + Right-click on the REST service and select New Resource.
     + Enter the resource path (e.g., / for all policies).
     + Right-click on the resource and select New Method.
     + Choose GET as the method.
     + Click on the method and then click Request.
     + Click the green play button to send the request and view the response.
   * **POST Request**: To create a new policy.
     + Right-click on the REST service and select New Resource.
     + Enter the resource path (e.g., /).
     + Right-click on the resource and select New Method.
     + Choose POST as the method.
     + Click on the method and then click Request.
     + In the request editor, set the Content-Type to application/json.
     + Enter the JSON body for the new policy (e.g., {"policyNumber":"12345","policyType":"Health","coverageDetails":"Full Coverage","contactInfo":"contact@example.com"}).
     + Click the green play button to send the request and view the response.
   * **PUT Request**: To update an existing policy.
     + Right-click on the REST service and select New Resource.
     + Enter the resource path (e.g., /{id}).
     + Right-click on the resource and select New Method.
     + Choose PUT as the method.
     + Click on the method and then click Request.
     + In the request editor, set the Content-Type to application/json.
     + Enter the JSON body for the updated policy (e.g., {"policyNumber":"12345","policyType":"Health","coverageDetails":"Full Coverage","contactInfo":"contact@example.com"}).
     + Click the green play button to send the request and view the response.
   * **DELETE Request**: To delete a policy.
     + Right-click on the REST service and select New Resource.
     + Enter the resource path (e.g., /{id}).
     + Right-click on the resource and select New Method.
     + Choose DELETE as the method.
     + Click on the method and then click Request.
     + Click the green play button to send the request and view the response.

**Step 5: Verify Responses**

1. **Check the response** for each request to ensure it matches the expected output.
2. **Use assertions** in SoapUI to automate response verification:
   * Click on the Assertions tab in the request editor.
   * Add assertions to check the response status, content, and other properties.

**Step 6: Automate Tests**

1. **Create test suites** to group related tests.
2. **Add test cases** to the test suite for each endpoint.
3. **Run the test suite** to execute all test cases and verify the results.

By following these steps, you can effectively test your Spring Boot application using SoapUI. If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Spring Boot Testing Tutorial - Crash Course](https://www.youtube.com/watch?v=aPoJPESMJBk)

[2] [Test your app for REAL: How to Integration Test in Spring Boot](https://www.youtube.com/watch?v=7QCzBwplNIk)

[3] [🔥🔥Unit Testing in Spring Boot Project in one video || Crash Course](https://www.youtube.com/watch?v=qpK1AoFWY8k)

[4] [Creating a SOAP Web Service with Spring - Baeldung](https://www.baeldung.com/spring-boot-soap-web-service)

[5] [How to automate API testing with SoapUi and Spring boot](https://stackoverflow.com/questions/59607233/how-to-automate-api-testing-with-soapui-and-spring-boot)

[6] [Spring Boot SOAP Webservice Example - HowToDoInJava](https://howtodoinjava.com/spring-boot/spring-boot-soap-webservice-example/)

[7] <https://github.com/devtiro/spring-boot-integration-test-tutorial>

[8] [https://www.devtiro.com](https://www.devtiro.com/)

[9] <https://www.instagram.com/devtiro>

[10] <http://www.w3.org/2001/XMLSchema>

[11] <http://www.baeldung.com/springsoap/gen>

[12] <http://localhost:8080/ws>

{

"policyNumber":"12345",

"policyType":"Health",

"coverageDetails":"Full Coverage",

"contactInfo":"contact@example.com"

}

To add OpenAPI to your Spring Boot project and access it via Swagger UI, follow these steps:

**Step 1: Add Dependencies**

Add the springdoc-openapi-starter-webmvc-ui dependency to your pom.xml.

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.0.0</version>

</dependency>

**Step 2: Configure Application Properties**

Add the following properties to your application.properties file to configure OpenAPI and Swagger UI.

springdoc.api-docs.path=/v3/api-docs

springdoc.swagger-ui.path=/swagger-ui.html

**Step 3: Create OpenAPI Configuration Class**

Create a configuration class to customize the OpenAPI documentation if needed.

package com.imsapp.ims.config;

import io.swagger.v3.oas.models.OpenAPI;

import io.swagger.v3.oas.models.info.Info;

import org.springdoc.core.customizers.OpenApiCustomiser;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class OpenApiConfig {

@Bean

public OpenAPI customOpenAPI() {

return new OpenAPI()

.info(new Info()

.title("Policy Management API")

.version("1.0")

.description("API for managing policies"));

}

@Bean

public OpenApiCustomiser sortOperations() {

return openApi -> openApi.getPaths().values().forEach(pathItem -> pathItem.readOperations().sort((o1, o2) -> o1.getOperationId().compareTo(o2.getOperationId())));

}

}

**Step 4: Access Swagger UI**

Run your Spring Boot application and access Swagger UI at http://localhost:8080/swagger-ui.html.

**Step 5: Verify Endpoints**

You should see all your API endpoints documented and accessible via Swagger UI. You can interact with the endpoints directly from the Swagger UI interface.

By following these steps, you can integrate OpenAPI and Swagger UI into your Spring Boot project, making it easier to document and test your API[[1]](https://www.baeldung.com/spring-rest-openapi-documentation)[[2]](https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api)[[3]](https://multigenesys.com/blog/integrating-swagger-with-openapi-in-spring-boot-a-comprehensive-guide). If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Documenting a Spring REST API Using OpenAPI 3.0 | Baeldung](https://www.baeldung.com/spring-rest-openapi-documentation)

[2] [Setting Up Swagger 2 with a Spring REST API | Baeldung](https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api)

[3] [How to use OpenAPI Swagger in Spring Boot guide - MultiGeneSys](https://multigenesys.com/blog/integrating-swagger-with-openapi-in-spring-boot-a-comprehensive-guide)

[4] [Spring Boot + Swagger 3 example (with OpenAPI 3) - BezKoder](https://www.bezkoder.com/spring-boot-swagger-3/)

[5] <http://localhost:8080/v3/api-docs>

[6] <http://localhost:8080/api-docs>

[7] <http://localhost:8080/api-docs.yaml>

[8] <http://localhost:8080/swagger-ui/index.html>

[9] <http://localhost:8080/swagger-ui-custom.html>

To set responses using OpenAPI annotations in your Spring Boot controller, you can use the @Operation and @ApiResponse annotations provided by the springdoc-openapi library. These annotations help document your API endpoints and their responses.

Here's how you can update your PolicyController to include OpenAPI annotations:

**Step 1: Add OpenAPI Annotations to Controller Methods**

Update your controller methods to include OpenAPI annotations.

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.PolicyRequest;

import com.imsapp.ims.dto.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.exception.PolicyNotFoundException;

import com.imsapp.ims.service.PolicyService;

import io.swagger.v3.oas.annotations.Operation;

import io.swagger.v3.oas.annotations.responses.ApiResponse;

import io.swagger.v3.oas.annotations.responses.ApiResponses;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/api/policies")

public class PolicyController {

@Autowired

private PolicyService policyService;

@Operation(summary = "Get all policies", description = "Retrieve a list of all policies")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully retrieved list"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@GetMapping

public List<PolicyResponse> getAllPolicies() {

return policyService.getAllPolicies().stream().map(this::convertToResponse).collect(Collectors.toList());

}

@Operation(summary = "Get policy by ID", description = "Retrieve a policy by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully retrieved policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@GetMapping("/{id}")

public ResponseEntity<PolicyResponse> getPolicyById(@PathVariable Long id) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

return ResponseEntity.ok(convertToResponse(policy));

}

@Operation(summary = "Create a new policy", description = "Create a new policy with the provided details")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully created policy"),

@ApiResponse(responseCode = "400", description = "Invalid input"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@PostMapping

public PolicyResponse createPolicy(@RequestBody PolicyRequest policyRequest) {

Policy policy = convertToEntity(policyRequest);

return convertToResponse(policyService.savePolicy(policy));

}

@Operation(summary = "Update an existing policy", description = "Update the details of an existing policy")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully updated policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "400", description = "Invalid input"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@PutMapping("/{id}")

public ResponseEntity<PolicyResponse> updatePolicy(@PathVariable Long id, @RequestBody PolicyRequest policyRequest) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return ResponseEntity.ok(convertToResponse(policyService.savePolicy(policy)));

}

@Operation(summary = "Delete a policy", description = "Delete a policy by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "204", description = "Successfully deleted policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@DeleteMapping("/{id}")

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

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policyService.deletePolicy(id);

return ResponseEntity.noContent().build();

}

private Policy convertToEntity(PolicyRequest policyRequest) {

Policy policy = new Policy();

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return policy;

}

private PolicyResponse convertToResponse(Policy policy) {

PolicyResponse policyResponse = new PolicyResponse();

policyResponse.setId(policy.getId());

policyResponse.setPolicyNumber(policy.getPolicyNumber());

policyResponse.setPolicyType(policy.getPolicyType());

policyResponse.setCoverageDetails(policy.getCoverageDetails());

policyResponse.setContactInfo(policy.getContactInfo());

return policyResponse;

}

}

**Step 2: Access Swagger UI**

Run your Spring Boot application and access Swagger UI at <http://localhost:8080/swagger-ui.html>. You should see all your API endpoints documented with the responses specified.

By following these steps, you can set responses using OpenAPI annotations in your controller and access them via Swagger UI[[1]](https://www.baeldung.com/spring-rest-openapi-documentation)[[2]](https://www.bezkoder.com/spring-boot-swagger-3/). If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Documenting a Spring REST API Using OpenAPI 3.0 | Baeldung](https://www.baeldung.com/spring-rest-openapi-documentation)

[2] [Spring Boot + Swagger 3 example (with OpenAPI 3) - BezKoder](https://www.bezkoder.com/spring-boot-swagger-3/)

[3] [Spring Boot Server Implementation Using OpenAPI Generator by Andriy Kucher](https://www.youtube.com/watch?v=FJv5sGt3EEs)

[4] [How To Add OpenAPI And Swagger To Spring Boot Application](https://www.youtube.com/watch?v=A_RWUcTqHBI)

[5] [Spring Boot tutorials | Swagger - Document your REST API using OpenAPI Specification (Swagger 3.0)](https://www.youtube.com/watch?v=v-kMcODmxYg)

[6] [Spring Boot Rest Controller how to return different HTTP status codes?](https://stackoverflow.com/questions/24292373/spring-boot-rest-controller-how-to-return-different-http-status-codes)

[7] <http://localhost:8080/v3/api-docs>

[8] <http://localhost:8080/swagger-ui.html>

[9] <http://localhost:8080/api-docs>

[10] <http://localhost:8080/api-docs.yaml>

[11] <http://localhost:8080/swagger-ui/index.html>

[12] <http://localhost:8080/swagger-ui-custom.html>

To handle validation at the attribute level of an entity in your Spring Boot project, you can use the Bean Validation API (Hibernate Validator is the reference implementation). Here's how you can add validation annotations to your Policy entity and handle validation errors globally.

**Step 1: Add Validation Dependencies**

Ensure you have the necessary validation dependencies in your pom.xml.

<dependency>

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

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

</dependency>

**Step 2: Add Validation Annotations to Entity**

Add validation annotations to your Policy entity.

package com.imsapp.ims.entity.policy;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.validation.constraints.NotBlank;

import jakarta.validation.constraints.Size;

import lombok.Getter;

import lombok.Setter;

@Entity

@Getter

@Setter

public class Policy {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "Policy number is mandatory")

@Size(max = 20, message = "Policy number must be less than 20 characters")

private String policyNumber;

@NotBlank(message = "Policy type is mandatory")

private String policyType;

@NotBlank(message = "Coverage details are mandatory")

private String coverageDetails;

@NotBlank(message = "Contact information is mandatory")

private String contactInfo;

}

**Step 3: Update Controller to Validate Request**

Use the @Valid annotation in your controller to validate the request body.

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.PolicyRequest;

import com.imsapp.ims.dto.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.exception.PolicyNotFoundException;

import com.imsapp.ims.service.PolicyService;

import io.swagger.v3.oas.annotations.Operation;

import io.swagger.v3.oas.annotations.responses.ApiResponse;

import io.swagger.v3.oas.annotations.responses.ApiResponses;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import jakarta.validation.Valid;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/api/policies")

public class PolicyController {

@Autowired

private PolicyService policyService;

@Operation(summary = "Get all policies", description = "Retrieve a list of all policies")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully retrieved list"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@GetMapping

public List<PolicyResponse> getAllPolicies() {

return policyService.getAllPolicies().stream().map(this::convertToResponse).collect(Collectors.toList());

}

@Operation(summary = "Get policy by ID", description = "Retrieve a policy by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully retrieved policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@GetMapping("/{id}")

public ResponseEntity<PolicyResponse> getPolicyById(@PathVariable Long id) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

return ResponseEntity.ok(convertToResponse(policy));

}

@Operation(summary = "Create a new policy", description = "Create a new policy with the provided details")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully created policy"),

@ApiResponse(responseCode = "400", description = "Invalid input"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@PostMapping

public PolicyResponse createPolicy(@Valid @RequestBody PolicyRequest policyRequest) {

Policy policy = convertToEntity(policyRequest);

return convertToResponse(policyService.savePolicy(policy));

}

@Operation(summary = "Update an existing policy", description = "Update the details of an existing policy")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Successfully updated policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "400", description = "Invalid input"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@PutMapping("/{id}")

public ResponseEntity<PolicyResponse> updatePolicy(@PathVariable Long id, @Valid @RequestBody PolicyRequest policyRequest) {

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return ResponseEntity.ok(convertToResponse(policyService.savePolicy(policy)));

}

@Operation(summary = "Delete a policy", description = "Delete a policy by its ID")

@ApiResponses(value = {

@ApiResponse(responseCode = "204", description = "Successfully deleted policy"),

@ApiResponse(responseCode = "404", description = "Policy not found"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@DeleteMapping("/{id}")

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

Policy policy = policyService.getPolicyById(id).orElseThrow(() -> new PolicyNotFoundException("Policy not found with id: " + id));

policyService.deletePolicy(id);

return ResponseEntity.noContent().build();

}

private Policy convertToEntity(PolicyRequest policyRequest) {

Policy policy = new Policy();

policy.setPolicyNumber(policyRequest.getPolicyNumber());

policy.setPolicyType(policyRequest.getPolicyType());

policy.setCoverageDetails(policyRequest.getCoverageDetails());

policy.setContactInfo(policyRequest.getContactInfo());

return policy;

}

private PolicyResponse convertToResponse(Policy policy) {

PolicyResponse policyResponse = new PolicyResponse();

policyResponse.setId(policy.getId());

policyResponse.setPolicyNumber(policy.getPolicyNumber());

policyResponse.setPolicyType(policy.getPolicyType());

policyResponse.setCoverageDetails(policy.getCoverageDetails());

policyResponse.setContactInfo(policy.getContactInfo());

return policyResponse;

}

}

**Step 4: Create Global Exception Handler for Validation Errors**

Create a global exception handler to handle validation errors.

package com.imsapp.ims.exception;

import com.imsapp.ims.dto.ErrorResponse;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.MethodArgumentNotValidException;

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

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

import org.springframework.web.context.request.WebRequest;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<ErrorResponse> handleValidationExceptions(MethodArgumentNotValidException ex, WebRequest request) {

Map<String, String> errors = new HashMap<>();

ex.getBindingResult().getFieldErrors().forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));

ErrorResponse errorResponse = new ErrorResponse(HttpStatus.BAD\_REQUEST, "Validation failed", request.getDescription(false));

errorResponse.setErrors(errors);

return new ResponseEntity<>(errorResponse, HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(PolicyNotFoundException.class)

public ResponseEntity<ErrorResponse> handlePolicyNotFoundException(PolicyNotFoundException ex, WebRequest request) {

ErrorResponse errorResponse = new ErrorResponse(HttpStatus.NOT\_FOUND, ex.getMessage(), request.getDescription(false));

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

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex, WebRequest request) {

ErrorResponse errorResponse = new ErrorResponse(HttpStatus.INTERNAL\_SERVER\_ERROR, ex.getMessage(), request.getDescription(false));

return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**Step 5: Update ErrorResponse Class**

Update the ErrorResponse class to include validation errors.

package com.imsapp.ims.dto;

import lombok.Getter;

import lombok.Setter;

import org.springframework.http.HttpStatus;

import java.time.LocalDateTime;

import java.util.Map;

@Getter

@Setter

public class ErrorResponse {

private LocalDateTime timestamp;

private HttpStatus status;

private String message;

private String path;

private Map<String, String> errors;

public ErrorResponse(HttpStatus status, String message, String path) {

this.timestamp = LocalDateTime.now();

this.status = status;

this.message = message;

this.path = path;

}

}

**Step 6: Test the Application**

Run your application and test the endpoints using tools like Postman or SoapUI to ensure validation works as expected.

By following these steps, you can handle validation at the attribute level of your entity and manage validation errors globally[[1]](https://www.baeldung.com/spring-boot-bean-validation)[[2]](https://stackoverflow.com/questions/62981169/how-to-validate-an-attribute-based-on-another-in-spring-boot-in-a-clean-way)[[3]](https://www.baeldung.com/spring-validator-interface). If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Validation in Spring Boot - Baeldung](https://www.baeldung.com/spring-boot-bean-validation)

[2] [java - How to validate an attribute based on another in spring-boot in ...](https://stackoverflow.com/questions/62981169/how-to-validate-an-attribute-based-on-another-in-spring-boot-in-a-clean-way)

[3] [Validation Using the Spring Validator Interface - Baeldung](https://www.baeldung.com/spring-validator-interface)

Adding SonarLint to your Spring Boot project helps you catch issues early in the development process. SonarLint is an IDE extension that provides on-the-fly feedback to developers on new bugs and quality issues injected into their code. Here’s how you can integrate SonarLint into your project:

**Step 1: Install SonarLint Plugin in Your IDE**

**For IntelliJ IDEA:**

1. **Open IntelliJ IDEA**.
2. Go to File -> Settings -> Plugins.
3. Search for "SonarLint" and install the plugin.
4. Restart IntelliJ IDEA.

**For Eclipse:**

1. **Open Eclipse**.
2. Go to Help -> Eclipse Marketplace.
3. Search for "SonarLint" and install the plugin.
4. Restart Eclipse.

**Step 2: Configure SonarLint**

**For IntelliJ IDEA:**

1. **Open IntelliJ IDEA**.
2. Go to File -> Settings -> Tools -> SonarLint.
3. Configure the connection to your SonarQube server if you have one, or use the default settings for local analysis.

**For Eclipse:**

1. **Open Eclipse**.
2. Go to Window -> Preferences -> SonarLint.
3. Configure the connection to your SonarQube server if you have one, or use the default settings for local analysis.

**Step 3: Analyze Your Code**

1. **Open your project** in your IDE.
2. SonarLint will automatically analyze your code and provide feedback on issues.
3. You can view the issues in the SonarLint tool window and fix them accordingly.

**Step 4: Integrate with SonarQube (Optional)**

If you want to integrate SonarLint with SonarQube for more comprehensive analysis:

1. **Set up SonarQube**:
   * Download and run SonarQube via Docker: docker run -d -p 9000:9000 sonarqube.
   * Access the SonarQube dashboard at [http://localhost:9000](http://localhost:9000/).
2. **Configure SonarQube in your IDE**:
   * In IntelliJ IDEA, go to File -> Settings -> Tools -> SonarLint -> Project Settings.
   * Add a connection to your SonarQube server.

**Step 5: Run SonarQube Analysis**

1. **Run Maven commands** to analyze your project and send results to SonarQube:

mvn clean install

mvn sonar:sonar

1. **Check the SonarQube dashboard** for detailed analysis results.

By following these steps, you can integrate SonarLint into your Spring Boot project and catch issues early in the development process[[1]](https://stackoverflow.com/questions/59011373/setting-up-properly-sonarqube-for-code-coverage)[[2]](https://dev.to/itscosmas/how-to-add-sonarqube-code-coverage-to-spring-boot-1lgk)[[3]](https://www.baeldung.com/sonar-qube). If you have any further questions or need additional assistance, feel free to ask!

**References**

[1] [Setting up properly SonarQube for Code Coverage](https://stackoverflow.com/questions/59011373/setting-up-properly-sonarqube-for-code-coverage)

[2] [How to add SonarQube Code Coverage to Spring Boot](https://dev.to/itscosmas/how-to-add-sonarqube-code-coverage-to-spring-boot-1lgk)

[3] [Code Analysis with SonarQube - Baeldung](https://www.baeldung.com/sonar-qube)

[4] [https://sonarcloud.io](https://sonarcloud.io/)

[5] <https://www.sonarsource.com/lp/products/sonarqube/static-code-analysis/>

[6] <https://central.sonatype.com/artifact/org.jacoco/jacoco-maven-plugin>

[7] <https://community.sonarsource.com/t/coverage-test-data-importing-jacoco-coverage-report-in-xml-format/12151>

[8] [http://localhost:9000](http://localhost:9000/)

Let's write JUnit test cases for the above Spring Boot application. We'll create both unit tests for the service layer and integration tests for the controller layer.

Sure! Let's write JUnit 5 test cases using Mockito for the PolicyService class in your Spring Boot 3 application.

Step 1: Add Testing Dependencies

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

Step 2: Create Unit Test Class for PolicyService

Create a test class for PolicyService in the src/test/java directory.

package com.imsapp.ims.service.policy;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.repository.policy.PolicyRepository;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

class PolicyServiceTest {

@Mock

private PolicyRepository policyRepository;

@InjectMocks

private PolicyService policyService;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testGetAllPolicies() {

Policy policy1 = new Policy();

policy1.setId(1L);

policy1.setPolicyNumber("12345");

policy1.setPolicyType("Health");

policy1.setCoverageDetails("Full Coverage");

policy1.setContactInfo("contact@example.com");

Policy policy2 = new Policy();

policy2.setId(2L);

policy2.setPolicyNumber("67890");

policy2.setPolicyType("Life");

policy2.setCoverageDetails("Partial Coverage");

policy2.setContactInfo("contact2@example.com");

List<Policy> policies = Arrays.asList(policy1, policy2);

when(policyRepository.findAll()).thenReturn(policies);

List<Policy> foundPolicies = policyService.getAllPolicies();

assertEquals(2, foundPolicies.size());

assertEquals("12345", foundPolicies.get(0).getPolicyNumber());

assertEquals("67890", foundPolicies.get(1).getPolicyNumber());

}

@Test

void testGetPolicyById() {

Policy policy = new Policy();

policy.setId(1L);

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyRepository.findById(1L)).thenReturn(Optional.of(policy));

Optional<Policy> foundPolicy = policyService.getPolicyById(1L);

assertEquals(1L, foundPolicy.get().getId());

assertEquals("12345", foundPolicy.get().getPolicyNumber());

}

@Test

void testSavePolicy() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyRepository.save(any(Policy.class))).thenReturn(policy);

Policy savedPolicy = policyService.savePolicy(policy);

verify(policyRepository, times(1)).save(policy);

assertEquals("12345", savedPolicy.getPolicyNumber());

}

@Test

void testDeletePolicy() {

doNothing().when(policyRepository).deleteById(1L);

policyService.deletePolicy(1L);

verify(policyRepository, times(1)).deleteById(1L);

}

}

Step 3: Run Your Tests

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your PolicyService class is working correctly. If you have any questions or need further assistance, feel free to ask!

References

**[1]** [**PolicyService**](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=e54f6d04-7c6b-43af-a846-9305f7a20cec&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMDI3NzkifQ.CgoKBHNuaWQSAjY0EgsI3vXrhrLN8j0QBRoNNDAuMTI2LjQ5LjE1MiosZUN2ZjlvZWZMcWFIZ2pwcHczbDFCbk9ZQzA0UEpKZGFncnpIckl2eEhxST0woQE4AUIQoYtVahugALD3o5gXytgoS0oQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hogEcbmlyYWphLmR3aXZlZGlAY2FwZ2VtaW5pLmNvbaoBEDEwMDMyMDAzNjEyNURCMDSyAS9hbGxmaWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.saJonwM1yJWb-cSBNclMIOD1rkBixOMILengE0w3-Jg&ApiVersion=2.0&web=1)

Let's write JUnit 5 test cases using Mockito for the PolicyController class to achieve full coverage. We'll create tests for all the endpoints in the controller.

Step 1: Add Testing Dependencies

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

Step 2: Create Unit Test Class for PolicyController

Create a test class for PolicyController in the src/test/java directory.

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.policy.PolicyRequest;

import com.imsapp.ims.dto.policy.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.exception.PolicyNotFoundException;

import com.imsapp.ims.service.policy.PolicyService;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

class PolicyControllerTest {

@Mock

private PolicyService policyService;

@InjectMocks

private PolicyController policyController;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testGetAllPolicies() {

Policy policy1 = new Policy();

policy1.setId(1L);

policy1.setPolicyNumber("12345");

policy1.setPolicyType("Health");

policy1.setCoverageDetails("Full Coverage");

policy1.setContactInfo("contact@example.com");

Policy policy2 = new Policy();

policy2.setId(2L);

policy2.setPolicyNumber("67890");

policy2.setPolicyType("Life");

policy2.setCoverageDetails("Partial Coverage");

policy2.setContactInfo("contact2@example.com");

List<Policy> policies = Arrays.asList(policy1, policy2);

when(policyService.getAllPolicies()).thenReturn(policies);

List<PolicyResponse> response = policyController.getAllPolicies();

assertEquals(2, response.size());

assertEquals("12345", response.get(0).getPolicyNumber());

assertEquals("67890", response.get(1).getPolicyNumber());

}

@Test

void testGetPolicyById() {

Policy policy = new Policy();

policy.setId(1L);

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyService.getPolicyById(1L)).thenReturn(Optional.of(policy));

ResponseEntity<PolicyResponse> response = policyController.getPolicyById(1L);

assertEquals(HttpStatus.OK, response.getStatusCode());

assertEquals("12345", response.getBody().getPolicyNumber());

}

@Test

void testGetPolicyById\_NotFound() {

when(policyService.getPolicyById(1L)).thenReturn(Optional.empty());

try {

policyController.getPolicyById(1L);

} catch (PolicyNotFoundException ex) {

assertEquals("Policy not found with id: 1", ex.getMessage());

}

}

@Test

void testCreatePolicy() {

PolicyRequest policyRequest = new PolicyRequest();

policyRequest.setPolicyNumber("12345");

policyRequest.setPolicyType("Health");

policyRequest.setCoverageDetails("Full Coverage");

policyRequest.setContactInfo("contact@example.com");

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyService.savePolicy(any(Policy.class))).thenReturn(policy);

PolicyResponse response = policyController.createPolicy(policyRequest);

assertEquals("12345", response.getPolicyNumber());

}

@Test

void testUpdatePolicy() {

PolicyRequest policyRequest = new PolicyRequest();

policyRequest.setPolicyNumber("12345");

policyRequest.setPolicyType("Health");

policyRequest.setCoverageDetails("Full Coverage");

policyRequest.setContactInfo("contact@example.com");

Policy policy = new Policy();

policy.setId(1L);

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyService.getPolicyById(1L)).thenReturn(Optional.of(policy));

when(policyService.savePolicy(any(Policy.class))).thenReturn(policy);

ResponseEntity<PolicyResponse> response = policyController.updatePolicy(1L, policyRequest);

assertEquals(HttpStatus.OK, response.getStatusCode());

assertEquals("12345", response.getBody().getPolicyNumber());

}

@Test

void testUpdatePolicy\_NotFound() {

PolicyRequest policyRequest = new PolicyRequest();

policyRequest.setPolicyNumber("12345");

policyRequest.setPolicyType("Health");

policyRequest.setCoverageDetails("Full Coverage");

policyRequest.setContactInfo("contact@example.com");

when(policyService.getPolicyById(1L)).thenReturn(Optional.empty());

try {

policyController.updatePolicy(1L, policyRequest);

} catch (PolicyNotFoundException ex) {

assertEquals("Policy not found with id: 1", ex.getMessage());

}

}

@Test

void testDeletePolicy() {

Policy policy = new Policy();

policy.setId(1L);

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyService.getPolicyById(1L)).thenReturn(Optional.of(policy));

doNothing().when(policyService).deletePolicy(1L);

ResponseEntity<Void> response = policyController.deletePolicy(1L);

assertEquals(HttpStatus.NO\_CONTENT, response.getStatusCode());

verify(policyService, times(1)).deletePolicy(1L);

}

@Test

void testDeletePolicy\_NotFound() {

when(policyService.getPolicyById(1L)).thenReturn(Optional.empty());

try {

policyController.deletePolicy(1L);

} catch (PolicyNotFoundException ex) {

assertEquals("Policy not found with id: 1", ex.getMessage());

}

}

}

Step 3: Run Your Tests

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your PolicyController class is working correctly and achieve full coverage. If you have any questions or need further assistance, feel free to ask!

References

[1] [PolicyController](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=f4118a5a-2d8e-4fe4-9a0e-b2ccb14f1fe4&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMDY4MTIifQ.CgoKBHNuaWQSAjY0EgsImp_DwN7P8j0QBRoNMjAuMTkwLjE2MC45OCosUlZNQ214Z0tKYWptUXBzVTd5K0pka2IwYTBPNTd2ek5iZ3REUVp3bTdWMD0woQE4AUIQoYtZQqHgAMAZ3QN9cERTGUoQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hogEcbmlyYWphLmR3aXZlZGlAY2FwZ2VtaW5pLmNvbaoBEDEwMDMyMDAzNjEyNURCMDSyAS9hbGxmaWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.6w2iI8cRoo4upvdyPyULi5_9y6BAis8AyWn7CMbg_gA&ApiVersion=2.0&web=1)

Writing test cases for all components of your application is a best practice to ensure that each part works correctly and integrates well with others. Here’s a breakdown of what you should consider testing:

**1. Service Layer**

**Why**: The service layer contains the business logic of your application. Testing this layer ensures that the logic is correct and handles various scenarios appropriately.

**How**: Use unit tests with Mockito to mock dependencies and test the service methods.

**Example**: Test cases for PolicyService.

**2. Controller Layer**

**Why**: The controller layer handles HTTP requests and responses. Testing this layer ensures that your endpoints work correctly and return the expected results.

**How**: Use integration tests with MockMvc to simulate HTTP requests and verify responses.

**Example**: Test cases for PolicyController.

**3. Repository Layer**

**Why**: The repository layer interacts with the database. Testing this layer ensures that your data access methods work correctly.

**How**: Use integration tests with an in-memory database (like H2) to test repository methods.

**Example**: Test cases for PolicyRepository.

**4. Request and Response DTOs**

**Why**: DTOs (Data Transfer Objects) are used to transfer data between layers. Testing these ensures that data is correctly mapped and validated.

**How**: Use unit tests to validate the correctness of DTOs and their mappings.

**Example**: Test cases for PolicyRequest and PolicyResponse.

**Comprehensive Testing Strategy**

1. **Unit Tests**: Focus on individual components (service methods, DTO validation).
2. **Integration Tests**: Focus on the interaction between components (controller endpoints, repository methods).
3. **End-to-End Tests**: Simulate real-world scenarios to ensure the entire application works as expected.

**Example Test Cases**

**Service Layer (PolicyServiceTest.java)**

package com.imsapp.ims.service.policy;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.repository.policy.PolicyRepository;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

class PolicyServiceTest {

@Mock

private PolicyRepository policyRepository;

@InjectMocks

private PolicyService policyService;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testGetPolicyById() {

Policy policy = new Policy();

policy.setId(1L);

when(policyRepository.findById(1L)).thenReturn(Optional.of(policy));

Optional<Policy> foundPolicy = policyService.getPolicyById(1L);

assertEquals(1L, foundPolicy.get().getId());

}

@Test

void testSavePolicy() {

Policy policy = new Policy();

when(policyRepository.save(any(Policy.class))).thenReturn(policy);

Policy savedPolicy = policyService.savePolicy(policy);

verify(policyRepository, times(1)).save(policy);

}

}

**Controller Layer (PolicyControllerTest.java)**

package com.imsapp.ims.controller.policy;

import com.imsapp.ims.dto.policy.PolicyRequest;

import com.imsapp.ims.dto.policy.PolicyResponse;

import com.imsapp.ims.entity.policy.Policy;

import com.imsapp.ims.exception.PolicyNotFoundException;

import com.imsapp.ims.service.policy.PolicyService;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.ArgumentMatchers.any;

import static org.mockito.Mockito.\*;

class PolicyControllerTest {

@Mock

private PolicyService policyService;

@InjectMocks

private PolicyController policyController;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testGetPolicyById() {

Policy policy = new Policy();

policy.setId(1L);

when(policyService.getPolicyById(1L)).thenReturn(Optional.of(policy));

ResponseEntity<PolicyResponse> response = policyController.getPolicyById(1L);

assertEquals(HttpStatus.OK, response.getStatusCode());

assertEquals("12345", response.getBody().getPolicyNumber());

}

@Test

void testCreatePolicy() {

PolicyRequest policyRequest = new PolicyRequest();

policyRequest.setPolicyNumber("12345");

policyRequest.setPolicyType("Health");

policyRequest.setCoverageDetails("Full Coverage");

policyRequest.setContactInfo("contact@example.com");

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

when(policyService.savePolicy(any(Policy.class))).thenReturn(policy);

PolicyResponse response = policyController.createPolicy(policyRequest);

assertEquals("12345", response.getPolicyNumber());

}

}

**Repository Layer (PolicyRepositoryTest.java)**

package com.imsapp.ims.repository.policy;

import com.imsapp.ims.entity.policy.Policy;

import org.junit.jupiter.api.Test;

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

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DataJpaTest

class PolicyRepositoryTest {

@Autowired

private PolicyRepository policyRepository;

@Test

void testFindById() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

policyRepository.save(policy);

Optional<Policy> foundPolicy = policyRepository.findById(policy.getId());

assertEquals("12345", foundPolicy.get().getPolicyNumber());

}

}

By writing test cases for all these components, you ensure comprehensive coverage and robustness of your application. If you have any further questions or need additional assistance, feel free to ask!

Let's write JUnit 5 test cases using Mockito for the Policy entity to achieve full coverage. Since the Policy class uses Lombok for getters and setters, we can focus on validating the constraints and ensuring that the getters and setters work correctly.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.2.0.Final</version>

</dependency>

<dependency>

<groupId>jakarta.validation</groupId>

<artifactId>jakarta.validation-api</artifactId>

<version>3.0.0</version>

</dependency>

**Step 2: Create Unit Test Class for Policy**

Create a test class for Policy in the src/test/java directory.

package com.imsapp.ims.entity.policy;

import jakarta.validation.ConstraintViolation;

import jakarta.validation.Validation;

import jakarta.validation.Validator;

import jakarta.validation.ValidatorFactory;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import java.util.Set;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertTrue;

class PolicyTest {

private Validator validator;

@BeforeEach

void setUp() {

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

validator = factory.getValidator();

}

@Test

void testPolicyValidation\_Success() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

Set<ConstraintViolation<Policy>> violations = validator.validate(policy);

assertTrue(violations.isEmpty());

}

@Test

void testPolicyValidation\_Failure() {

Policy policy = new Policy();

policy.setPolicyNumber(""); // Invalid: Blank

policy.setPolicyType(""); // Invalid: Blank

policy.setCoverageDetails(""); // Invalid: Blank

policy.setContactInfo(""); // Invalid: Blank

Set<ConstraintViolation<Policy>> violations = validator.validate(policy);

assertEquals(4, violations.size());

}

@Test

void testPolicyNumberSizeValidation() {

Policy policy = new Policy();

policy.setPolicyNumber("123456789012345678901"); // Invalid: More than 20 characters

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

Set<ConstraintViolation<Policy>> violations = validator.validate(policy);

assertEquals(1, violations.size());

assertEquals("Policy number must be less than 20 characters", violations.iterator().next().getMessage());

}

@Test

void testGettersAndSetters() {

Policy policy = new Policy();

policy.setId(1L);

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

assertEquals(1L, policy.getId());

assertEquals("12345", policy.getPolicyNumber());

assertEquals("Health", policy.getPolicyType());

assertEquals("Full Coverage", policy.getCoverageDetails());

assertEquals("contact@example.com", policy.getContactInfo());

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your Policy entity is correctly validated and that the constraints are enforced. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [Policy 1](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=061fb78a-46c6-4963-912d-674c4690aadd&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTU2MDYifQ.CgoKBHNuaWQSAjY0EgsIgIT02O3U8j0QBRoNNDAuMTI2LjMyLjE2MSosZ2xiUjU1bCtFOHRnSE1nMlBlaWJmN0xpVXIxalVJakg5Y2djTTlYOXFWQT0woQE4AUIQoYthpZJgALD3o5JWptZz70oQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hogEcbmlyYWphLmR3aXZlZGlAY2FwZ2VtaW5pLmNvbaoBEDEwMDMyMDAzNjEyNURCMDSyAS9hbGxmaWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.tzCRLIrMrnGGy6WZ6OMzWIKVPhcJRDqZ-OEo2Wp-KzQ&ApiVersion=2.0&web=1)

Let's write JUnit 5 test cases using Mockito for the OpenApiConfig class. Since this class primarily involves configuration and bean creation, we will focus on verifying that the customOpenAPI method correctly creates and configures the OpenAPI bean.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Test Class for OpenApiConfig**

Create a test class for OpenApiConfig in the src/test/java directory.

package com.imsapp.ims.config;

import io.swagger.v3.oas.models.OpenAPI;

import io.swagger.v3.oas.models.info.Info;

import org.junit.jupiter.api.Test;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertNotNull;

class OpenApiConfigTest {

@Test

void testCustomOpenAPI() {

OpenApiConfig openApiConfig = new OpenApiConfig();

OpenAPI openAPI = openApiConfig.customOpenAPI();

assertNotNull(openAPI);

assertNotNull(openAPI.getInfo());

assertEquals("Insurance Management System (IMS)", openAPI.getInfo().getTitle());

assertEquals("1.0", openAPI.getInfo().getVersion());

assertEquals("API for managing policies", openAPI.getInfo().getDescription());

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your OpenApiConfig class is correctly creating and configuring the OpenAPI bean. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [OpenApiConfig](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=60f13cbc-7111-4b00-905e-deb5ca0f83e9&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTYxODIifQ.CgoKBHNuaWQSAjY0EgsIjsqw05jV8j0QBRoNMjAuMTkwLjE2MC4yNiosR1FJWFBDeTFiK1NLTDMwSnp5Q0VhTlpGOVhaL0YrYWZlcFpielpvSXN6MD0woQE4AUIQoYtiMlQwALD3o5ccYWVhYUoQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hogEcbmlyYWphLmR3aXZlZGlAY2FwZ2VtaW5pLmNvbaoBEDEwMDMyMDAzNjEyNURCMDSyAS9hbGxmaWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.7LPArAanTXGFi4Eyf5BDsOKvDKDBVfz0rXl5zbRRq_o&ApiVersion=2.0&web=1)

Let's write JUnit 5 test cases using Mockito for the ErrorResponse class. Since this class is a simple data holder with getters and setters, we will focus on verifying that the getters and setters work correctly and that the constructor initializes the fields as expected.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Test Class for ErrorResponse**

Create a test class for ErrorResponse in the src/test/java directory.

package com.imsapp.ims.exception;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.springframework.http.HttpStatus;

import java.time.LocalDateTime;

import java.util.HashMap;

import java.util.Map;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertNotNull;

class ErrorResponseTest {

private ErrorResponse errorResponse;

private LocalDateTime timestamp;

private HttpStatus status;

private String message;

private String path;

private Map<String, String> errors;

@BeforeEach

void setUp() {

timestamp = LocalDateTime.now();

status = HttpStatus.BAD\_REQUEST;

message = "Validation failed";

path = "/api/policies";

errors = new HashMap<>();

errors.put("policyNumber", "must not be blank");

errorResponse = new ErrorResponse(status, message, path);

errorResponse.setErrors(errors);

}

@Test

void testErrorResponseInitialization() {

assertNotNull(errorResponse);

assertEquals(status, errorResponse.getStatus());

assertEquals(message, errorResponse.getMessage());

assertEquals(path, errorResponse.getPath());

assertNotNull(errorResponse.getTimestamp());

}

@Test

void testGettersAndSetters() {

errorResponse.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR);

errorResponse.setMessage("Internal server error");

errorResponse.setPath("/api/errors");

errorResponse.setErrors(null);

assertEquals(HttpStatus.INTERNAL\_SERVER\_ERROR, errorResponse.getStatus());

assertEquals("Internal server error", errorResponse.getMessage());

assertEquals("/api/errors", errorResponse.getPath());

assertEquals(null, errorResponse.getErrors());

}

@Test

void testErrors() {

assertEquals(errors, errorResponse.getErrors());

assertEquals("must not be blank", errorResponse.getErrors().get("policyNumber"));

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

Let's write JUnit 5 test cases using Mockito for the PolicyNotFoundException class. Since this class is a simple custom exception, we will focus on verifying that the exception is correctly instantiated and that the message is properly set.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Test Class for PolicyNotFoundException**

Create a test class for PolicyNotFoundException in the src/test/java directory.

package com.imsapp.ims.exception;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertThrows;

class PolicyNotFoundExceptionTest {

@Test

void testPolicyNotFoundExceptionMessage() {

String errorMessage = "Policy not found with id: 1";

PolicyNotFoundException exception = assertThrows(PolicyNotFoundException.class, () -> {

throw new PolicyNotFoundException(errorMessage);

});

assertEquals(errorMessage, exception.getMessage());

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your PolicyNotFoundException class is correctly instantiated and that the message is properly set. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [PolicyNotFoundException](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=99c9b9ec-468a-4704-8712-abf39931991e&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTY1MDcifQ.CgoKBHNuaWQSAjY0EgsI-Kim7rDV8j0QBRoMNDAuMTI2LjMyLjk5KixpUzBHUENRQklpUkhYSmJxWVdjRjdVVkxpdFRjMjQ4STYwUytXWllvZC84PTChATgBQhChi2KBqkAAwBndBVEdwHJ7ShBoYXNoZWRwcm9vZnRva2VuUh1bImttc2kiLCJkdmNfY21wIiwiZHZjX2RtamQiXXIpMGguZnxtZW1iZXJzaGlwfDEwMDMyMDAzNjEyNWRiMDRAbGl2ZS5jb216ATKCARIJWq6idgCfa08Rle1dM9d8TWGiARxuaXJhamEuZHdpdmVkaUBjYXBnZW1pbmkuY29tqgEQMTAwMzIwMDM2MTI1REIwNLIBL2FsbGZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkyAEB.MptyHB8TY8vTZkr2CbdmfHrS7CqRO_RdHGawHknEF54&ApiVersion=2.0&web=1)

This setup will help you ensure that your ErrorResponse class is correctly initialized and that the getters and setters work as expected. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [ErrorResponse](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=bfe8d3a8-3c22-43b3-b1f4-047dc35e02a2&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTYzNjIifQ.CgoKBHNuaWQSAjY0EgsIqMzfgqbV8j0QBRoOMjAuMTkwLjE2MC4xNjMqLE1ncFRScUN5S21hUml5U1FpNFBxaXpYME5OSksxUWY5aGdNTG0rTmU1WmM9MKEBOAFCEKGLYl4i4ADAGd0An8WKFwxKEGhhc2hlZHByb29mdG9rZW5SHVsia21zaSIsImR2Y19jbXAiLCJkdmNfZG1qZCJdcikwaC5mfG1lbWJlcnNoaXB8MTAwMzIwMDM2MTI1ZGIwNEBsaXZlLmNvbXoBMoIBEglarqJ2AJ9rTxGV7V0z13xNYaIBHG5pcmFqYS5kd2l2ZWRpQGNhcGdlbWluaS5jb22qARAxMDAzMjAwMzYxMjVEQjA0sgEvYWxsZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWTIAQE.eWE3g_SqNZYUAA5y5Hhojkg5HSvA6UzP6mLqzb9OkPo&ApiVersion=2.0&web=1)

Let's write JUnit 5 test cases using Mockito for the GlobalExceptionHandler class. We will focus on testing the exception handling methods to ensure they return the correct responses.

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.0.0</version>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Test Class for GlobalExceptionHandler**

Create a test class for GlobalExceptionHandler in the src/test/java directory.

package com.imsapp.ims.exception;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.MethodArgumentNotValidException;

import org.springframework.web.context.request.WebRequest;

import org.springframework.web.context.request.async.WebAsyncManager;

import java.util.HashMap;

import java.util.Map;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

class GlobalExceptionHandlerTest {

@InjectMocks

private GlobalExceptionHandler globalExceptionHandler;

@Mock

private WebRequest webRequest;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void testHandlePolicyNotFoundException() {

PolicyNotFoundException ex = new PolicyNotFoundException("Policy not found with id: 1");

when(webRequest.getDescription(false)).thenReturn("/api/policies/1");

ResponseEntity<ErrorResponse> response = globalExceptionHandler.handlePolicyNotFoundException(ex, webRequest);

assertEquals(HttpStatus.NOT\_FOUND, response.getStatusCode());

assertEquals("Policy not found with id: 1", response.getBody().getMessage());

assertEquals("/api/policies/1", response.getBody().getPath());

}

@Test

void testHandleGlobalException() {

Exception ex = new Exception("Internal server error");

when(webRequest.getDescription(false)).thenReturn("/api/policies");

ResponseEntity<ErrorResponse> response = globalExceptionHandler.handleGlobalException(ex, webRequest);

assertEquals(HttpStatus.INTERNAL\_SERVER\_ERROR, response.getStatusCode());

assertEquals("Internal server error", response.getBody().getMessage());

assertEquals("/api/policies", response.getBody().getPath());

}

@Test

void testHandleValidationExceptions() {

MethodArgumentNotValidException ex = mock(MethodArgumentNotValidException.class);

when(webRequest.getDescription(false)).thenReturn("/api/policies");

when(ex.getBindingResult().getFieldErrors()).thenReturn(List.of(

new FieldError("policy", "policyNumber", "must not be blank"),

new FieldError("policy", "policyType", "must not be blank")

));

ResponseEntity<ErrorResponse> response = globalExceptionHandler.handleValidationExceptions(ex, webRequest);

assertEquals(HttpStatus.BAD\_REQUEST, response.getStatusCode());

assertEquals("Validation failed", response.getBody().getMessage());

assertEquals("/api/policies", response.getBody().getPath());

assertEquals(2, response.getBody().getErrors().size());

assertEquals("must not be blank", response.getBody().getErrors().get("policyNumber"));

assertEquals("must not be blank", response.getBody().getErrors().get("policyType"));

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your GlobalExceptionHandler class correctly handles exceptions and returns the appropriate responses. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [GlobalExceptionHandler](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=362b71de-e816-4eee-8ab3-977fbba2ea0d&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTY2MzAifQ.CgoKBHNuaWQSAjY0EgsIgquTgLrV8j0QBRoMNDAuMTI2LjMyLjk5KixRZm44a21DNVcxUmdJcldRMXE1ZDdqMXJGK29ycDJhY05xVlVjbFpEMjg0PTChATgBQhChi2KfngAAwBndD9m8OK1iShBoYXNoZWRwcm9vZnRva2VuUh1bImttc2kiLCJkdmNfY21wIiwiZHZjX2RtamQiXXIpMGguZnxtZW1iZXJzaGlwfDEwMDMyMDAzNjEyNWRiMDRAbGl2ZS5jb216ATKCARIJWq6idgCfa08Rle1dM9d8TWGiARxuaXJhamEuZHdpdmVkaUBjYXBnZW1pbmkuY29tqgEQMTAwMzIwMDM2MTI1REIwNLIBL2FsbGZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkyAEB.UNUCgd7WfFZmWdvrPy2M14iBdJr5BTNa6BJaFZeNmTE&ApiVersion=2.0&web=1)

Let's generate JUnit test cases for the PolicyRepository class. Since this class is an interface extending JpaRepository, we will focus on testing the repository methods using an in-memory database (H2).

**Step 1: Add Testing Dependencies**

Ensure you have the necessary testing dependencies in your pom.xml.

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>test</scope>

</dependency>

**Step 2: Create Unit Test Class for PolicyRepository**

Create a test class for PolicyRepository in the src/test/java directory.

package com.imsapp.ims.repository.policy;

import com.imsapp.ims.entity.policy.Policy;

import org.junit.jupiter.api.Test;

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

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;

import org.springframework.test.context.ActiveProfiles;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertTrue;

@DataJpaTest

@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)

@ActiveProfiles("test")

class PolicyRepositoryTest {

@Autowired

private PolicyRepository policyRepository;

@Test

void testSavePolicy() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

Policy savedPolicy = policyRepository.save(policy);

assertEquals("12345", savedPolicy.getPolicyNumber());

assertEquals("Health", savedPolicy.getPolicyType());

assertEquals("Full Coverage", savedPolicy.getCoverageDetails());

assertEquals("contact@example.com", savedPolicy.getContactInfo());

}

@Test

void testFindById() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

Policy savedPolicy = policyRepository.save(policy);

Optional<Policy> foundPolicy = policyRepository.findById(savedPolicy.getId());

assertTrue(foundPolicy.isPresent());

assertEquals("12345", foundPolicy.get().getPolicyNumber());

}

@Test

void testFindAll() {

Policy policy1 = new Policy();

policy1.setPolicyNumber("12345");

policy1.setPolicyType("Health");

policy1.setCoverageDetails("Full Coverage");

policy1.setContactInfo("contact@example.com");

Policy policy2 = new Policy();

policy2.setPolicyNumber("67890");

policy2.setPolicyType("Life");

policy2.setCoverageDetails("Partial Coverage");

policy2.setContactInfo("contact2@example.com");

policyRepository.save(policy1);

policyRepository.save(policy2);

Iterable<Policy> policies = policyRepository.findAll();

assertEquals(2, ((Collection<?>) policies).size());

}

@Test

void testDeleteById() {

Policy policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

Policy savedPolicy = policyRepository.save(policy);

policyRepository.deleteById(savedPolicy.getId());

Optional<Policy> foundPolicy = policyRepository.findById(savedPolicy.getId());

assertTrue(foundPolicy.isEmpty());

}

}

**Step 3: Run Your Tests**

You can run your tests using your IDE or by running the following Maven command:

mvn test

This setup will help you ensure that your PolicyRepository class is correctly interacting with the database and performing CRUD operations as expected. If you have any questions or need further assistance, feel free to ask!

**References**

[1] [PolicyRepository](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=fa3dcd5c-9542-4e52-9d8e-f232ce0d8b94&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTY4MzAifQ.CgoKBHNuaWQSAjY0EgsIsIH98sjV8j0QBRoNMjAuMTkwLjE2MC45OCosT3c4KzZETnpBQVR6Wk9udWQ3aVpXZHMrc0N6alN1aGdLVFJaMUROa1ZQND0woQE4AUIQoYti0G2QALD3o5BoHdm0mkoQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hogEcbmlyYWphLmR3aXZlZGlAY2FwZ2VtaW5pLmNvbaoBEDEwMDMyMDAzNjEyNURCMDSyAS9hbGxmaWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.9fiVtk73LOSEb9XCK_lBaOHuXBwEwzzIe8T6sRkDhTg&ApiVersion=2.0&web=1)

Sure, I can help with that! Here's a test case for the PolicyRequest class:

package com.imsapp.ims.dto.policy;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class PolicyRequestTest {

@Test

public void testPolicyRequest() {

PolicyRequest policyRequest = new PolicyRequest();

// Test setting and getting policyNumber

policyRequest.setPolicyNumber("12345");

assertEquals("12345", policyRequest.getPolicyNumber());

// Test setting and getting policyType

policyRequest.setPolicyType("Health");

assertEquals("Health", policyRequest.getPolicyType());

// Test setting and getting coverageDetails

policyRequest.setCoverageDetails("Full Coverage");

assertEquals("Full Coverage", policyRequest.getCoverageDetails());

// Test setting and getting contactInfo

policyRequest.setContactInfo("contact@example.com");

assertEquals("contact@example.com", policyRequest.getContactInfo());

}

}

This test case uses JUnit 5 to verify that the getter and setter methods of the PolicyRequest class are working correctly. If you need any further assistance or modifications, feel free to ask!

**References**

[1] [PolicyRequest 1](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=5e66e2d5-e959-41e0-904f-7f8275d3154f&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTcyOTgifQ.CgoKBHNuaWQSAjY0EgsIuqq-5evV8j0QBRoOMjAuMTkwLjE3Ny4xNTIqLGowWi9iZUp4alFtUE5lVjE4Z2NrUUU5RVVWZjJJYkp4VDZDczNIa3RTd0E9MKEBOAFCEKGLY0LFUADAGd0N1j6KNKJKEGhhc2hlZHByb29mdG9rZW5SHVsia21zaSIsImR2Y19jbXAiLCJkdmNfZG1qZCJdcikwaC5mfG1lbWJlcnNoaXB8MTAwMzIwMDM2MTI1ZGIwNEBsaXZlLmNvbXoBMoIBEglarqJ2AJ9rTxGV7V0z13xNYaIBHG5pcmFqYS5kd2l2ZWRpQGNhcGdlbWluaS5jb22qARAxMDAzMjAwMzYxMjVEQjA0sgEvYWxsZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWTIAQE._DfbznAva3ABm3iIC4o0bxURlpJ82oAvHfevF9hXx54&ApiVersion=2.0&web=1)

Sure, I can help with that! Here's a JUnit test case for the Policy class using Mockito and Spring Boot Starter Validation:

package com.imsapp.ims.entity.policy;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.junit.jupiter.MockitoExtension;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.validation.BeanPropertyBindingResult;

import org.springframework.validation.Validator;

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

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.test.context.ContextConfiguration;

import jakarta.validation.Validation;

import jakarta.validation.ValidatorFactory;

import static org.junit.jupiter.api.Assertions.\*;

@ExtendWith(MockitoExtension.class)

@SpringBootTest

@ContextConfiguration(classes = {Policy.class})

public PolicyTest {

@InjectMocks

private Policy policy;

@Autowired

private Validator validator;

@BeforeEach

public void setUp() {

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

validator = factory.getValidator();

}

@Test

public void testPolicyValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertFalse(bindingResult.hasErrors());

}

@Test

public void testPolicyNumberValidation() {

policy = new Policy();

policy.setPolicyNumber(""); // Invalid policy number

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy number is mandatory", bindingResult.getFieldError("policyNumber").getDefaultMessage());

}

@Test

public void testPolicyTypeValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType(""); // Invalid policy type

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy type is mandatory", bindingResult.getFieldError("policyType").getDefaultMessage());

}

@Test

public void testCoverageDetailsValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails(""); // Invalid coverage details

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Coverage details are mandatory", bindingResult.getFieldError("coverageDetails").getDefaultMessage());

}

@Test

public void testContactInfoValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo(""); // Invalid contact info

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Contact information is mandatory", bindingResult.getFieldError("contactInfo").getDefaultMessage());

}

}

This test case uses Mockito for dependency injection and Spring Boot Starter Validation to validate the fields of the Policy class. Each test method checks for specific validation constraints and ensures that the validation messages are correct. If you need any further assistance or modifications, feel free to ask!

**References**

[1] [Policy 2](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=ab33377d-7098-4754-91b2-b2ae3a1cc19e&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTc3ODUifQ.CgoKBHNuaWQSAjY0EgsI1t_jiZDW8j0QBRoOMjAuMTkwLjE2MC4xNjMqLHNJYXBuTnd5eGJKNDVxOUR5MnhPaFdtODJ0ejdyZENkS211WkxWWGxraUU9MKEBOAFCEKGLY7mmQADAGd0FUb1paFdKEGhhc2hlZHByb29mdG9rZW5SHVsia21zaSIsImR2Y19jbXAiLCJkdmNfZG1qZCJdcikwaC5mfG1lbWJlcnNoaXB8MTAwMzIwMDM2MTI1ZGIwNEBsaXZlLmNvbXoBMoIBEglarqJ2AJ9rTxGV7V0z13xNYaIBHG5pcmFqYS5kd2l2ZWRpQGNhcGdlbWluaS5jb22qARAxMDAzMjAwMzYxMjVEQjA0sgEvYWxsZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkIGFsbHByb2ZpbGVzLnJlYWTIAQE.oGKK_ZOglItwSCs_aXKIe25IkeEZjbtkMHpzhJdObyo&ApiVersion=2.0&web=1)

Sure, here's a JUnit test case for the Policy class using Spring Boot Starter Validation, JUnit 5, and Mockito:

package com.imsapp.ims.entity.policy;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.junit.jupiter.MockitoExtension;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.validation.BeanPropertyBindingResult;

import org.springframework.validation.Validator;

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

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.test.context.ContextConfiguration;

import jakarta.validation.Validation;

import jakarta.validation.ValidatorFactory;

import static org.junit.jupiter.api.Assertions.\*;

@ExtendWith(MockitoExtension.class)

@SpringBootTest

@ContextConfiguration(classes = {Policy.class})

public class PolicyTest {

@InjectMocks

private Policy policy;

@Autowired

private Validator validator;

@BeforeEach

public void setUp() {

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

validator = factory.getValidator();

}

@Test

public void testPolicyValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertFalse(bindingResult.hasErrors());

}

@Test

public void testPolicyNumberValidation() {

policy = new Policy();

policy.setPolicyNumber(""); // Invalid policy number

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy number is mandatory", bindingResult.getFieldError("policyNumber").getDefaultMessage());

}

@Test

public void testPolicyTypeValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType(""); // Invalid policy type

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy type is mandatory", bindingResult.getFieldError("policyType").getDefaultMessage());

}

@Test

public void testCoverageDetailsValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails(""); // Invalid coverage details

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Coverage details are mandatory", bindingResult.getFieldError("coverageDetails").getDefaultMessage());

}

@Test

public void testContactInfoValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo(""); // Invalid contact info

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Contact information is mandatory", bindingResult.getFieldError("contactInfo").getDefaultMessage());

}

}

This test case uses Mockito for dependency injection and Spring Boot Starter Validation to validate the fields of the Policy class. Each test method checks for specific validation constraints and ensures that the validation messages are correct. If you need any further assistance or modifications, feel free to ask!

**References**

[1] [Policy 3](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=6ccc45a5-e2b4-432e-b6c2-e405e24852a7&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTc5NjQifQ.CgoKBHNuaWQSAjY0EgsIvsCrtZ3W8j0QBRoMNDAuMTI2LjMyLjk5Kix5dEx2Yk1xNGx0bXl5aVNtSkowTnpTa3dtTVRaNlhmWEhMMmM4ZFdhbGtJPTChATgBQhChi2PlYNAAsPejnLVwC-ShShBoYXNoZWRwcm9vZnRva2VuUh1bImttc2kiLCJkdmNfY21wIiwiZHZjX2RtamQiXXIpMGguZnxtZW1iZXJzaGlwfDEwMDMyMDAzNjEyNWRiMDRAbGl2ZS5jb216ATKCARIJWq6idgCfa08Rle1dM9d8TWGiARxuaXJhamEuZHdpdmVkaUBjYXBnZW1pbmkuY29tqgEQMTAwMzIwMDM2MTI1REIwNLIBL2FsbGZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZCBhbGxwcm9maWxlcy5yZWFkyAEB.G9RVUlJ0Xt6MqSOrTUlhoTfeMDcQRfsDZLMDfHVUW4s&ApiVersion=2.0&web=1)

The error indicates that the Jakarta Bean Validation provider (such as Hibernate Validator) is missing from your classpath. To resolve this issue, you need to add the Hibernate Validator dependency to your project. Here's how you can do it:

1. **Add Hibernate Validator Dependency**: If you're using Maven, add the following dependency to your pom.xml:

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.2.0.Final</version>

</dependency>

If you're using Gradle, add the following dependency to your build.gradle:

implementation 'org.hibernate.validator:hibernate-validator:6.2.0.Final'

1. **Update Your Test Class**: Ensure that your test class is correctly configured to use the Validator bean. Here's the updated version of your PolicyTest class:

package com.imsapp.ims.entity.policy;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.junit.jupiter.MockitoExtension;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.validation.BeanPropertyBindingResult;

import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.test.context.junit.jupiter.SpringExtension;

import jakarta.validation.Validation;

import jakarta.validation.ValidatorFactory;

import static org.junit.jupiter.api.Assertions.\*;

@ExtendWith({MockitoExtension.class, SpringExtension.class})

@SpringBootTest

public class PolicyTest {

@InjectMocks

private Policy policy;

@Autowired

private org.springframework.validation.Validator validator;

@BeforeEach

public void setUp() {

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

validator = new LocalValidatorFactoryBean();

((LocalValidatorFactoryBean) validator).afterPropertiesSet();

}

@Test

public void testPolicyValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertFalse(bindingResult.hasErrors());

}

@Test

public void testPolicyNumberValidation() {

policy = new Policy();

policy.setPolicyNumber(""); // Invalid policy number

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy number is mandatory", bindingResult.getFieldError("policyNumber").getDefaultMessage());

}

@Test

public void testPolicyTypeValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType(""); // Invalid policy type

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Policy type is mandatory", bindingResult.getFieldError("policyType").getDefaultMessage());

}

@Test

public void testCoverageDetailsValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails(""); // Invalid coverage details

policy.setContactInfo("contact@example.com");

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Coverage details are mandatory", bindingResult.getFieldError("coverageDetails").getDefaultMessage());

}

@Test

public void testContactInfoValidation() {

policy = new Policy();

policy.setPolicyNumber("12345");

policy.setPolicyType("Health");

policy.setCoverageDetails("Full Coverage");

policy.setContactInfo(""); // Invalid contact info

BeanPropertyBindingResult bindingResult = new BeanPropertyBindingResult(policy, "policy");

validator.validate(policy, bindingResult);

assertTrue(bindingResult.hasErrors());

assertEquals("Contact information is mandatory", bindingResult.getFieldError("contactInfo").getDefaultMessage());

}

@Configuration

static class TestConfig {

@Bean

public org.springframework.validation.Validator validator() {

return new LocalValidatorFactoryBean();

}

}

}

After adding the Hibernate Validator dependency and updating your test class, the ApplicationContext should load successfully, and your tests should run without errors. If you need any further assistance or modifications, feel free to ask!

**References**

[1] [PolicyTest](https://capgemini-my.sharepoint.com/personal/niraja_dwivedi_capgemini_com/_layouts/15/download.aspx?UniqueId=2db0fae4-fb64-408b-a92f-59419d56dec8&Translate=false&tempauth=v1.eyJzaXRlaWQiOiIxN2ZhYzM1Ni00OTY3LTQ2ZWEtOWEzMC1jN2FhMDQxZjUwYWIiLCJhcHBfZGlzcGxheW5hbWUiOiJPZmZpY2UgMzY1IFNlYXJjaCBTZXJ2aWNlIiwiYXBwaWQiOiI2NmE4ODc1Ny0yNThjLTRjNzItODkzYy0zZThiZWQ0ZDY4OTkiLCJhdWQiOiIwMDAwMDAwMy0wMDAwLTBmZjEtY2UwMC0wMDAwMDAwMDAwMDAvY2FwZ2VtaW5pLW15LnNoYXJlcG9pbnQuY29tQDc2YTJhZTVhLTlmMDAtNGY2Yi05NWVkLTVkMzNkNzdjNGQ2MSIsImV4cCI6IjE3NDIyMTkxMzgifQ.CgoKBHNuaWQSAjY0EgsIzpXA7vTW8j0QBRoNMjAuMTkwLjE2MC45NyoseEE4c3Nnb2xXK1lGZDZ5ZUFJdXJNSTFDK1RZZnVNQlplL0dVVXFGWTFmOD0woQE4AUIQoYtlA-yAAMAZ3QVF-7GcY0oQaGFzaGVkcHJvb2Z0b2tlblIdWyJrbXNpIiwiZHZjX2NtcCIsImR2Y19kbWpkIl1yKTBoLmZ8bWVtYmVyc2hpcHwxMDAzMjAwMzYxMjVkYjA0QGxpdmUuY29tegEyggESCVquonYAn2tPEZXtXTPXfE1hkgEGTmlyYWphmgEHRHdpdmVkaaIBHG5pcmFqYS5kd2l2ZWRpQGNhcGdlbWluaS5jb22qARAxMDAzMjAwMzYxMjVEQjA0sgE6YWxsZmlsZXMud3JpdGUgbXlmaWxlcy53cml0ZSBteWZpbGVzLnJlYWQgYWxscHJvZmlsZXMucmVhZMgBAQ.h2jfmIBzaqNC7vvNYL_x65Jr5OrRCQm2tBRFj7pe638&ApiVersion=2.0&web=1)

The NullPointerException is occurring because the getBindingResult() method of the MethodArgumentNotValidException mock is returning null. To fix this, you need to mock the BindingResult and set it up properly. Here's the corrected test case:

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.BindingResult;

import org.springframework.validation.FieldError;

import org.springframework.web.bind.MethodArgumentNotValidException;

import org.springframework.web.context.request.WebRequest;

import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

public class GlobalExceptionHandlerTest {

private final GlobalExceptionHandler globalExceptionHandler = new GlobalExceptionHandler();

private final WebRequest webRequest = mock(WebRequest.class);

@Test

void testHandleValidationExceptions() {

// Mock the MethodArgumentNotValidException and BindingResult

MethodArgumentNotValidException ex = mock(MethodArgumentNotValidException.class);

BindingResult bindingResult = mock(BindingResult.class);

// Set up the mock behavior

when(webRequest.getDescription(false)).thenReturn("/api/policies");

when(ex.getBindingResult()).thenReturn(bindingResult);

when(bindingResult.getFieldErrors()).thenReturn(List.of(

new FieldError("policy", "policyNumber", "must not be blank"),

new FieldError("policy", "policyType", "must not be blank")

));

// Call the method under test

ResponseEntity<ErrorResponse> response = globalExceptionHandler.handleValidationExceptions(ex, webRequest);

// Assert the response

assertEquals(HttpStatus.BAD\_REQUEST, response.getStatusCode());

assertEquals("Validation failed", response.getBody().getMessage());

assertEquals("/api/policies", response.getBody().getPath());

assertEquals(2, response.getBody().getErrors().size());

assertEquals("must not be blank", response.getBody().getErrors().get("policyNumber"));

assertEquals("must not be blank", response.getBody().getErrors().get("policyType"));

}

}

In this corrected version, I've added a mock for the BindingResult and set it up to return the list of FieldError objects. This should resolve the NullPointerException and allow your test to run successfully. If you need any further assistance or modifications, feel free to ask!