**Ques: - Interceptor in spring boot**

**Ans: -**

* An interceptor is a component that can be used to intercept HTTP requests and responses.
* They are commonly used for tasks such as logging, authentication, performance monitoring, or modifying requests and responses before or after the actual controller methods are invoked.

**Types of Interceptors in Spring Boot**

1. **HandlerInterceptor: -**

* The **HandlerInterceptor** interface in Spring allows us to intercept the request before it reaches the controller or after the controller has processed the request but before the view is rendered.

**Steps to Implement a HandlerInterceptor:**

1. **Create a custom interceptor**:

* Implement the HandlerInterceptor interface
* Override the preHandle(), postHandle(), and afterCompletion() methods.

1. **Register the Interceptor**

* Register the custom interceptor in the Spring configuration.

**Example of a custom HandlerInterceptor**

import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class MyCustomInterceptor implements HandlerInterceptor {

// Before the controller method is invoked

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

// Log request details or perform authentication checks

System.out.println("Request intercepted before controller method: " + request.getRequestURI());

return true; // Return true to allow the request to proceed to the controller

}

// After the controller method is invoked but before the view is rendered

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

// Modify the model or log after controller method execution

System.out.println("Request intercepted after controller method: " + request.getRequestURI());

}

// After the complete request has been processed

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

// Perform any cleanup activities

System.out.println("Request processing complete: " + request.getRequestURI());

}

}

1. **Registering the Interceptor: -**

To register the custom interceptor, you need to implements WebMvcConfigurer and override the addInterceptors() method.

import org.springframework.context.annotation.Configuration;

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

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

@Configuration

public class WebConfig implements WebMvcConfigurer {

@Override

public void addInterceptors(InterceptorRegistry registry) {

// Register your interceptor

registry.addInterceptor(new MyCustomInterceptor())

.addPathPatterns("/api/\*\*") // Apply to specific URL patterns

.excludePathPatterns("/api/public/\*\*"); // Exclude specific URL patterns

}

}

**2.Filter in Spring Boot**

* Filters in Spring Boot are another way to handle HTTP requests and responses.
* Filters are more flexible but operate at the Servlet level, not at the Spring MVC level.

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.annotation.WebFilter;

import java.io.IOException;

**@WebFilter(filterName = "myFilter", urlPatterns = "/api/\*")**

public class MyFilter **implements Filter {**

**@Override**

public void **init(**FilterConfig filterConfig**) throws ServletException {**

**// Initialization logic, if needed**

**}**

**@Override**

public void **doFilter(**ServletRequest request, ServletResponse response, FilterChain chain**)**

**throws IOException, ServletException {**

**System.out.println("**Filter applied before request is passed to the controller.**");**

**chain.doFilter**(request, response); **// Continue the request-response chain**

**System.out.println("**Filter applied after response is returned.**");**

**}**

**@Override**

**public void destroy() {**

// Cleanup logic, if needed

**}**

**}**

**import org.springframework.boot.web.servlet.FilterRegistrationBean;**

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

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

**@Configuration**

public class FilterConfig {

@Bean

public FilterRegistrationBean<MyFilter> loggingFilter() {

FilterRegistrationBean<MyFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new MyFilter());

registrationBean.addUrlPatterns("/api/\*"); // Apply to specific URL patterns

return registrationBean;

}

}

**Key Differences Between Interceptors and Filters:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **HandlerInterceptor** | **Filter** |
| **Scope** | Works only within Spring MVC controllers | Works only within Spring MVC controllers |
| **Execution Order** | Executes before/after controller methods | Executes before the request reaches the Servlet or after the response leaves it |
| **Use Case** | Ideal for controller-specific logic | Ideal for tasks like logging, security, CORS, etc |
| **Access to Spring MVC** | Has access to the ModelAndView | No access to ModelAndView |

**Conclusion:**

* Interceptors are a great choice when you need to intercept and modify HTTP requests before or after they reach the Spring MVC controller. They allow more integration with Spring's web components like models and views.
* Filters are more general-purpose and can be used for broader use cases like logging, security checks, and modifying request/response headers, working at the Servlet level.

**Both techniques are powerful and can be used depending on your needs in a Spring Boot application.**

**Ques: - Parallel stream vs Sequential stream**

**Ans: -**

Sequential Stream

* A sequential stream processes elements one at a time, in a single-threaded manner:

Example:

import java.util.Arrays;

import java.util.List;

public class SequentialStreamExample {

public static void main(String[] args) {

List<String> myList = Arrays.asList("a", "b", "c", "d", "e");

myList.stream()

.forEach(**ele** -> System.out.println(Thread.currentThread().getName() + " - " + **ele**));

}

}

Output:

main - a

main - b

main - c

main - d

main - e

Here, each element is processed sequentially in the main thread.

Parallel Stream

* A parallel stream processes elements concurrently, utilizing multiple threads:

Example:

import java.util.Arrays;

import java.util.List;

public class ParallelStreamExample {

public static void main(String[] args) {

List<String> myList = Arrays.asList("a", "b", "c", "d", "e");

myList.parallelStream()

.forEach(**ele** -> System.out.println(Thread.currentThread().getName() + " - " + **ele**));

}

}

Output:

ForkJoinPool.commonPool-worker-1 - a

main - b

ForkJoinPool.commonPool-worker-2 - c

ForkJoinPool.commonPool-worker-3 - d

ForkJoinPool.commonPool-worker-1 – e

**Key Differences**

* **Threading:** Sequential streams use a single thread, while parallel streams use multiple threads.
* **Order:** Sequential streams maintain the order of elements, whereas parallel streams may not.
* **Performance:** Parallel streams can improve performance for large datasets by leveraging multiple CPU cores, but they also introduce overhead due to thread management.

**Ques: - map vs flat map**

**Ans: -**

**map**

* map is used to transform each element of the stream by applying a function to it.
* The function applied to each element returns a single value which can be of the same type or a different type.

Example

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

class Product {

String name;

Product(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

public class MapExample {

public static void main(String[] args) {

List<Product> products = Arrays.asList(

new Product("Laptop"),

new Product("Phone"),

new Product("Tablet")

);

List<String> productNames = products.stream()

.map(Product::getName)

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.println(productNames); // Output: [LAPTOP, PHONE, TABLET]

}

}

**flatMap**

* flatMap is used to transform each element of the stream by applying a function that returns a stream of values, and then flattening those multiple streams into a single stream.

Example

1. **let's assume we have a list of accounts,**
2. **each account has multiple transactions.**
3. **We want to flatten this nested structure to get a list of all transactions across all accounts.**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

class Transaction {

String id;

Transaction(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

class Account {

List<Transaction> transactions;

Account(List<Transaction> transactions) {

this.transactions = transactions;

}

public List<Transaction> getTransactions() {

return transactions;

}

}

public class FlatMapExample {

public static void main(String[] args) {

List<Account> accounts = Arrays.asList(

new Account(Arrays.asList(new Transaction("T1"), new Transaction("T2"))),

new Account(Arrays.asList(new Transaction("T3"), new Transaction("T4"))),

new Account(Arrays.asList(new Transaction("T5")))

);

List<Transaction> allTransactions = accounts.stream()

.flatMap(account -> account.getTransactions().stream())

.collect(Collectors.toList());

allTransactions.forEach(transaction -> System.out.println(transaction.getId()));

// Output: T1, T2, T3, T4, T5

}

}

**Key Takeaways**

* **map:** Used for simple transformations where each element is converted individually.
* **flatMap:** Used for flattening nested structures, transforming each element into a stream and then merging these streams into one.

**Ques: - return type of map**

**Ans:**

**The return type of map in Java varies depending on the context in which it is used:**

**Stream<T> Context**

* When used with a Stream<T>, the return type of map is Stream<R>
* where R is the result type of applying the mapping function to the elements of type T

Example:

Stream<String> **stringStream** = Stream.of("1", "2", "3");

Stream<Integer> integerStream = **stringStream.map**(Integer::valueOf);

**Input Type:** Stream<String>

**Return Type:** Stream<Integer>

Optional<T> Context

* When used with an Optional<T>, the return type of map is Optional<R>
* where R is the result type of applying the mapping function to the value inside the Optional.

Example:

Optional<String> optionalString = Optional.of("42");

Optional<Integer> optionalInteger = optionalString.map(Integer::valueOf);

**Input Type:** Optional<String>

**Return Type:** Optional<Integer>

**Ques: - mock vs spy vs injectmock**

**Ans: -**

**@Mock**

* @Mock is used to create and inject mock objects.
* A mock object simulates the behavior of a real object in a controlled way, ideal for isolating the class being tested.

**Example:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

class ServiceTest {

@Mock

private Dependency dependency;

@Test

void testService() {

MockitoAnnotations.openMocks(this);

**when(dependency.doSomething()).thenReturn("Mocked Response");**

Service service = new Service(dependency);

String result = service.performTask();

assertEquals("Mocked Response", result);

}

}

**Note: - Here, @Mock creates a mock instance of Dependency which is used in the Service class.**

**@Spy**

* @Spy is used to create a spy object.
* A spy wraps around an existing instance and allows you to call real methods while selectively mocking others.
* It’s useful when you want to partially mock a class.

**Example:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Spy;

import org.mockito.MockitoAnnotations;

class ServiceTest {

@Spy

private Dependency dependency = new Dependency();

@Test

void testService() {

MockitoAnnotations.openMocks(this);

**doReturn("Mocked Response").when(dependency).doSomething();**

Service service = new Service(dependency);

String result = service.performTask();

assertEquals("Mocked Response", result);

}

}

**Note: @Spy wraps the real Dependency object, allowing us to call its real methods while mocking doSomething().**

**@InjectMocks**

* @InjectMocks is used to automatically inject mock or spy instances into the class being tested. It simplifies dependency injection for unit tests.

**Example:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

class ServiceTest {

@Mock

private Dependency dependency;

@InjectMocks

private Service service;

@Test

void testService() {

MockitoAnnotations.openMocks(this);

when(dependency.doSomething()).thenReturn("Mocked Response");

String result = **service.performTask();**

assertEquals("Mocked Response", result);

}

}

**Note: @InjectMocks injects the mocked Dependency into the Service instance, streamlining the setup process.**

**Key Differences**

* **@Mock:** Creates mock instances, ideal for isolating class behavior.
* **@Spy**: Partially mocks an object, allowing real method calls while overriding specific behavior.
* **@InjectMocks:** Automatically injects mock or spy instances into the class under test.

**Ques:. what are different state of circuit breaker and how to implement on above example?**

**Ans:.**

The circuit breaker pattern has three primary states:

1. **Closed State**:

* **Normal Operation**: Requests flow through the circuit breaker to the service as usual.
* **Failure Count**: The circuit breaker monitors the number of failures.
* **Transition to Open**: If the number of consecutive failures exceeds a predefined threshold, the circuit breaker transitions to the Open state.

1. **Open State**:

* **Failure Mode**: Requests are immediately blocked or rejected, returning an error to the client.
* **Timeout for Retry**: After a specified timeout, the circuit breaker allows a limited number of test requests to pass through to check if the service has recovered.

1. **Half-Open State**:

* **Test Mode**: A limited number of requests are allowed through to test the service.
* **Transition to Closed**: If the test requests are successful, the circuit breaker transitions back to the Closed state.
* **Transition to Open**: If the test requests fail, the circuit breaker transitions back to the Open state.

**Ques:. For fault tolerance, use Resilience4j, explain configuration in detail.**

**Ans:.**

spring:

cloud:

gateway:

routes:

- id: stock\_service

uri: lb://stock-service

predicates:

- Path=/stock/\*\*

filters:

- name: JwtAuthenticationFilter

- name: CircuitBreaker

args:

name: stockServiceCircuitBreaker

fallbackUri: forward:/fallback/stock

resilience4j:

circuitbreaker:

configs:

default:

register-health-indicator: true

sliding-window-size: 100

sliding-window-type: COUNT\_BASED

minimum-number-of-calls: 10

wait-duration-in-open-state: 10000

failure-rate-threshold: 50

record-exceptions:

- java.io.IOException

- java.util.concurrent.TimeoutException

instances:

stockServiceCircuitBreaker:

base-config: default

**Resilience4j Circuit Breaker Configuration**

1. **register-health-indicator: true**
   * **Purpose**: This setting enables the registration of a health indicator for the circuit breaker with Spring Boot's health endpoint (/actuator/health).
   * **Benefit**: This allows you to monitor the state of the circuit breaker via Spring Boot Actuator, making it easier to observe the health and status of your circuit breakers.
2. **sliding-window-size: 100**
   * **Purpose**: This specifies the size of the sliding window that is used to record the outcome of calls (successes and failures).
   * **COUNT\_BASED**: The sliding window can be based on count or time.
     + **COUNT\_BASED**: The window records the outcome of the last N calls.
     + **TIME\_BASED**: The window records the outcomes of calls within the last X seconds.
   * **Benefit**: A larger window size provides a more stable measure of the error rate, while a smaller window size makes the circuit breaker more responsive to changes in the error rate.
3. **minimum-number-of-calls: 10**
   * **Purpose**: This is the minimum number of calls required in the sliding window before the circuit breaker can calculate the error rate and potentially open the circuit.
   * **Benefit**: This prevents the circuit breaker from opening prematurely when there are very few calls, ensuring that the error rate calculation is based on a sufficient sample size.
4. **wait-duration-in-open-state: 10000**
   * **Purpose**: This is the amount of time (in milliseconds) that the circuit breaker stays open before transitioning to the half-open state to test if the backend has recovered.
   * **Benefit**: This setting helps in preventing repeated requests to a failing service, allowing it some time to recover before test calls are allowed again.
5. **failure-rate-threshold: 50**
   * **Purpose**: This sets the threshold for the failure rate (as a percentage) at which the circuit breaker will open.
   * **Benefit**: If more than 50% of the calls in the sliding window fail, the circuit breaker will open, preventing further calls to the failing service and triggering fallback mechanisms.
6. **record-exceptions:**
   * **java.io.IOException**
   * **java.util.concurrent.TimeoutException**
   * **Purpose**: This specifies the types of exceptions that should be recorded as failures by the circuit breaker.
   * **Benefit**: This allows you to define which exceptions are considered failures. For example, IOException and TimeoutException indicate network or timeout issues, which are critical for determining the service's reliability.

**Summary**

* **Health Indicator**: Enables monitoring of circuit breaker health.
* **Sliding Window**: A count-based window of the last 100 calls is used to calculate the error rate.
* **Minimum Calls**: At least 10 calls are needed in the sliding window for the circuit breaker to calculate the error rate.
* **Open State Duration**: The circuit breaker remains open for 10 seconds before allowing test requests.
* **Failure Rate Threshold**: If 50% or more of the calls fail, the circuit breaker opens.
* **Recorded Exceptions**: Defines which exceptions are counted as failures.