Spring Cloud Circuit Breaker: Step-by-Step Tutorial

author: Ramesh Fadatare

SPRING BOOT SPRING CLOUD

In a microservices architecture, failure is inevitable. A single failing service can cause a cascading effect, leading to system-wide failures. To prevent such scenarios, implementing circuit breakers is crucial. Spring Cloud Circuit Breaker provides a powerful and easy-to-use abstraction for integrating various circuit breaker implementations into your microservices.

In this tutorial, we will set up a Spring Boot application that uses Spring Cloud Circuit Breaker to handle service failures gracefully. We will demonstrate its real use by creating two microservices (`ProductService` and `OrderService`) and applying a circuit breaker to the `OrderService` when it calls the `ProductService`.

Prerequisites

- JDK 17 or higher
- Maven or Gradle
- IDE (e.g., IntelliJ IDEA, Eclipse)
- Basic knowledge of Spring Boot

What is a Circuit Breaker?

A circuit breaker is a design pattern used in microservices architecture to prevent cascading failures and provide fallback mechanisms. When a service call fails repeatedly, the circuit breaker trips and directs the calls to a fallback method, thereby preventing further failures and allowing the system to recover gracefully.

Key Benefits

1. **Fault Tolerance**: Prevents cascading failures by stopping the flow of requests to a failing service.
2. **Fallback Mechanisms**: Provides alternate responses when a service fails.
3. **Resilience**: Enhances the overall resilience of the system by isolating failures.

Setting Up Spring Cloud Circuit Breaker

Step 1: Create `ProductService` Microservice

1.1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
   - **Group**: `com.example`
   - **Artifact**: `product-service`
   - **Name**: `Product Service`

    - **Description**: `E-commerce Product Service`
    - **Package Name**: `com.example.productservice`
    - **Packaging**: `Jar`
    - **Java**: `17` (or higher)
3. Add dependencies:
    - **Spring Web**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

1.2: Define the Product Model

Create a new class `Product` in `src/main/java/com/example/productservice`:

```java
package com.example.productservice;

public class Product {
    private Long id;
    private String name;
    private double price;

    // Constructors, getters, and setters
    public Product() {}

    public Product(Long id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}
```

1.3: Create a ProductController

Create a new class `ProductController` in `src/main/java/com/example/productservice`:

```java
package com.example.productservice;

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

@RestController
public class ProductController {

    @GetMapping("/products/{id}")
    public Product getProductById(@PathVariable Long id) {
        // For simplicity, returning a hardcoded product. In a real application, you'd query the database.
        return new Product(id, "Sample Product", 99.99);
    }
}
```

1.4: Run `ProductService`

Ensure the main application class is set up correctly:

```java
package com.example.productservice;

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

@SpringBootApplication
public class ProductServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}
```

Step 2: Create `OrderService` Microservice

2.1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
    - **Group**: `com.example`
    - **Artifact**: `order-service`
    - **Name**: `Order Service`

    - **Description**: `E-commerce Order Service`
    - **Package Name**: `com.example.orderservice`
    - **Packaging**: `Jar`
    - **Java**: `17` (or higher)
3. Add dependencies:
    - **Spring Web**
    - **Spring Cloud Circuit Breaker**
    - **Spring Boot Starter Actuator** (for monitoring)
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

2.2: Add Spring Cloud Circuit Breaker Dependency

Add the following dependency to your `pom.xml`:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

2.3: Define the Order Model

Create a new class `Order` in `src/main/java/com/example/orderservice`:

```java
package com.example.orderservice;

public class Order {
    private Long id;
    private String product;
    private int quantity;

    // Constructors, getters, and setters
    public Order() {}

    public Order(Long id, String product, int quantity) {
        this.id = id;
        this.product = product;
        this.quantity = quantity;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getProduct() {
        return product;
    }

    public void setProduct(String product) {
        this.product = product;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

2.4: Create a ProductServiceClient

Create a new class `ProductServiceClient` in `src/main/java/com/example/orderservice` to call the `ProductService`:

```
package com.example.orderservice;

import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ProductServiceClient {

    private final RestTemplate restTemplate;

    public ProductServiceClient(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public Product getProductById(Long id) {
        return restTemplate.getForObject("http://localhost:8081/products/" + id, Product.class);
    }
}
```

2.5: Create a RestTemplate Bean

Add a `RestTemplate` bean to the `OrderService` application:

```
package com.example.orderservice;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

2.6: Create an OrderController with Circuit Breaker

Create a new class `OrderController` in `src/main/java/com/example/orderservice` and use Spring Cloud Circuit Breaker:

```java
package com.example.orderservice;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    private final ProductServiceClient productServiceClient;

    public OrderController(ProductServiceClient productServiceClient) {
        this.productServiceClient = productServiceClient;
    }

    @CircuitBreaker(name = "productService", fallbackMethod = "fallbackGetProductById")
    @GetMapping("/orders/{id}")
    public Order getOrderById(@PathVariable Long id) {
        Product product = productServiceClient.getProductById(id);
        return new Order(id, product.getName(), 2);
    }

    public Order fallbackGetProductById(Long id, Throwable throwable) {
        return new Order(id, "Fallback Product", 0);
    }
}
```

Step 3: Run and Test the Microservices

1. **Start `ProductService`**:
    - Run the `ProductServiceApplication` main class.
    - Verify it is running by accessing `http://localhost:8081/products/1` in your browser. You should see a sample product response.
2. **Start `OrderService`**:
    - Run the `OrderServiceApplication` main class.
    - Verify it is running by accessing `http://localhost:8082/orders/1` in your browser. You should see an order response with product details.

3. **Simulate a Failure**:

   - Stop the `ProductService`.
   - Access `http://localhost:8082/orders/1` in your browser. You should see the fallback response with the "Fallback Product" and quantity 0.

Conclusion

Spring Cloud Circuit Breaker provides a robust and easy-to-use solution for handling service failures gracefully. By following this tutorial, you've learned how to set up two microservices (`ProductService` and `OrderService`) and apply a circuit breaker to manage failures in the `OrderService` when calling the `ProductService`. This setup enhances the resilience and fault tolerance of your microservices architecture, ensuring that failures are handled gracefully and the overall system remains stable.