Spring Cloud API Gateway Global Filter Example - Spring Boot Microservices

author: Ramesh Fadatare

MICROSERVICES SPRING BOOT SPRING CLOUD

In this tutorial, we will create a Spring Cloud API Gateway project and demonstrate how to implement a global filter. We will use Spring Boot 3.2 and Spring Cloud 2023.x versions. The global filter will be applied to all incoming requests passing through the API Gateway.

Prerequisites

Before we start, ensure you have the following:

- Java Development Kit (JDK) installed
- Apache Maven installed
- An IDE (such as IntelliJ IDEA, Eclipse, or VS Code) installed

Step 1: Setting Up the Spring Cloud API Gateway Project

1.1 Create a Spring Cloud API Gateway Project

1. **Open Spring Initializr**:
    - Go to **Spring Initializr** in your web browser.
2. **Configure Project Metadata**:
    - **Project**: Maven Project
    - **Language**: Java
    - **Spring Boot**: Select the latest version of Spring Boot 3.2
    - **Group**: com.example
    - **Artifact**: api-gateway
    - **Name**: api-gateway
    - **Description**: Spring Cloud API Gateway Example
    - **Package Name**: com.example.apigateway
    - **Packaging**: Jar
    - **Java Version**: 17 (or your preferred version)
    - Click `Next`.
3. **Select Dependencies**:
    - On the `Dependencies` screen, select the dependencies you need:
        - Spring Cloud Gateway

        - Spring Boot DevTools
        - Spring Web
    - Click `Next`.

4. **Generate the Project**:
    - Click `Generate` to download the project zip file.
    - Extract the zip file to your desired location.

5. **Open the Project in Your IDE**:
    - Open your IDE and import the project as a Maven project.

1.2 Update `pom.xml`

Ensure your `pom.xml` includes the Spring Cloud dependencies. Add the Spring Cloud BOM (Bill of Materials) for the 2023.x version to manage the dependencies:

```
<properties>
    <java.version>17</java.version>
    <spring-cloud.version>2023.0.0</spring-cloud.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
```

```
    </dependency>
</dependencies>
```

Step 2: Configure the Application Properties

2.1 Update `application.yml`

Create an `application.yml` file in the `src/main/resources` directory and configure it as follows:

```yaml
server:
  port: 8080

spring:
  application:
    name: api-gateway

  cloud:
    gateway:
      routes:
        - id: example_service
          uri: http://localhost:8081
          predicates:
            - Path=/example/**
```

This configuration sets up a route that forwards requests matching `/example/**` to a service running on `http://localhost:8081`.

Step 3: Implementing a Global Filter

3.1 Create the `GlobalFilter` Class

Create a new class named `CustomGlobalFilter` in the `com.example.apigateway.filter` package:

```java
package com.example.apigateway.filter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;

import org.springframework.web.server.ServerWebExchange;

@Component
@Order(0)
public class CustomGlobalFilter implements GlobalFilter {

    private static final Logger logger = LoggerFactory.getLogger(CustomGlobalFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, org.springframework.cloud.gateway.filter.GatewayFilterChain chai
```

```
        logger.info("Global Filter executed for request: {}", exchange.getRequest().getPath());
        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
            logger.info("Global Filter post-processing for response: {}", exchange.getResponse().getStatusCode());
        }));
    }
}
```

Explanation:

- The `CustomGlobalFilter` class implements the `GlobalFilter` interface.
- The `filter` method logs the request path and response status code.
- The `@Component` annotation registers the filter as a Spring bean.
- The `@Order(0)` annotation ensures this filter runs first among multiple filters.

Step 4: Running the Application

4.1 Create the `ApiGatewayApplication` Class

Ensure the `ApiGatewayApplication` class is present in the `src/main/java/com/example/apigateway` directory:

```
package com.example.apigateway;

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

@SpringBootApplication
public class ApiGatewayApplication {

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

4.2 Run the Application

1. Open the `ApiGatewayApplication` class in the `src/main/java/com/example/apigateway` directory.
2. Click the green `Run` button in your IDE or use the terminal to run the application:

   ```
   ./mvnw spring-boot:run
   ```

3. The application will start on `http://localhost:8080`.

Step 5: Testing the Application

5.1 Test the Global Filter

1. Start the service running on `http://localhost:8081` that the API Gateway routes to. This can be any simple Spring Boot application that handles requests at the `/example` endpoint.

2. Use a tool like Postman to send a request to `http://localhost:8080/example/test`.
3. Check the logs to verify that the global filter has been executed. You should see log entries indicating that the global filter processed the request and response.

Conclusion

In this tutorial, we created a Spring Cloud API Gateway project using Spring Boot 3.2 and Spring Cloud 2023.x versions. We configured the application to route requests to a backend service and implemented a global filter to log requests and responses. This setup provides a solid foundation for developing more complex API Gateway functionalities.