Spring Cloud Gateway Tutorial

author: Ramesh Fadatare

SPRING BOOT SPRING CLOUD

Managing multiple services and their interactions can become complex and challenging in a microservices architecture. This is where an API Gateway comes into play, acting as a single entry point for all client requests. Spring Cloud Gateway is a powerful and flexible framework designed to build such gateways, providing a range of features to simplify and enhance your microservices interactions. In this blog post, we'll create two simple microservices and use Spring Cloud Gateway to route requests to these services. We will use the latest version of the libraries and test our setup with Postman.

Prerequisites

- JDK 17 or higher
- Maven or Gradle
- IDE (e.g., IntelliJ IDEA, Eclipse)
- Postman client

What is Spring Cloud Gateway?

Spring Cloud Gateway is a library for building API gateways on top of the Spring ecosystem. It provides a simple, effective way to route to APIs and provides cross-cutting concerns such as security, monitoring/metrics, and resiliency. Spring Cloud Gateway aims to provide a better, non-blocking alternative to traditional API gateway solutions, utilizing modern technology such as Spring WebFlux to handle network traffic asynchronously and ensure scalability.

Key Features of Spring Cloud Gateway:

Routing: Spring Cloud Gateway includes the ability to route API calls transparently to various backend services based on request attributes (like paths and headers). This routing is configured through routes that can be statically defined in configuration files or dynamically through a database.

Cross-Cutting Concerns: It handles cross-cutting concerns such as security, monitoring, and resiliency. This includes the integration of rate limiting, authentication and authorization, and response modifications.

Reactive Stack: Built on the reactive programming model using Spring WebFlux, it supports non-blocking API patterns. This is particularly useful for handling large concurrent connections with minimal resources.

Filter Capability: Filters allow the gateway to modify requests and responses before they are sent or returned. Filters in Spring Cloud Gateway can perform various actions, such as modifying request headers or aggregating multiple requests into a single response.

Integration: Easily integrates with other Spring Cloud projects for discovery and configuration. For example, it can dynamically utilize Spring Cloud DiscoveryClient to route requests based on service discovery.

Setting Up Spring Cloud Gateway

We will create two microservices: `ProductService` and `OrderService`. These services will provide product and order-related operations.

Create ProductService

Step 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.

Step 2: Define the Product Model

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

```
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;
    }
}
```

Step 3: Create a ProductController

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

```
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);
    }
}
```

Step 4: Configure the Server Port 8081

Open the application.properties file and add the following property to it:

server.port=8081

Step 5: Run ProductService

Ensure the main application class is set up correctly:

```
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);
    }
}
```

Create OrderService

Step 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**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

Step 2: Define the Order Model

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

```
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;
    }
}
```

Step 3: Create an OrderController

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

```
package com.example.orderservice;

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 {

    @GetMapping("/orders/{id}")
    public Order getOrderById(@PathVariable Long id) {
        // For simplicity, returning a hardcoded order. In a real application, you'd query the database.
        return new Order(id, "Sample Product", 2);
    }
}
```

Step 4: Configure the Server Port 8082

Open the application.properties file and add the following property to it:

server.port=8082

Step 5: Run OrderService

Ensure the main application class is set up correctly:

```java
package com.example.orderservice;

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

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

Create the Spring Cloud Gateway Application

Step 1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
    - **Group**: `com.example`
    - **Artifact**: `gateway-service`
    - **Name**: `Gateway Service`
    - **Description**: `API Gateway using Spring Cloud Gateway`
    - **Package Name**: `com.example.gatewayservice`
    - **Packaging**: `Jar`
    - **Java**: `11` (or higher)
3. Add dependencies:
    - **Spring Boot Starter Web**
    - **Spring Cloud Gateway**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

Step 2: Configure the Gateway

Create an `application.yml` file in the `src/main/resources` directory:

```yaml
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: http://localhost:8081
          predicates:
            - Path=/products/**
          filters:
            - StripPrefix=1
        - id: order-service
          uri: http://localhost:8082
```

```
        predicates:
          - Path=/orders/**
        filters:
          - StripPrefix=1
```

Explanation:

- `spring.cloud.gateway.routes`: Defines the routes for the gateway.
  - `id`: A unique identifier for the route.
  - `uri`: The URI of the backend service.
  - `predicates`: Conditions that must be met for the route to be applied.
  - `filters`: Modifications are to be applied to the request or response.

Step 3: Run GatewayService

Ensure the main application class is set up correctly:

```
package com.example.gatewayservice;

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

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

Step 3: Test with Postman

1. **Start All Services**: Run `ProductService`, `OrderService`, and `GatewayService` applications. Note that ProductService is running on port 8081, OrderService on port 8082, and GatewayService on port 8080.
2. **Open Postman**: Use Postman to send requests to the API Gateway.
3. The **API Gateway** handles requests and dynamically routes them to different backend APIs based on the route definitions.

Test Product Service

- **URL**: `http://localhost:8080/products/1`
- **Method**: GET
- **Expected Response**:

```
{
  "id": 1,
  "name": "Sample Product",
  "price": 99.99
}
```

Test Order Service

- **URL**: `http://localhost:8080/orders/1`
- **Method**: GET
- **Expected Response**:

```
{
  "id": 1,
  "product": "Sample Product",
  "quantity": 2
}
```

Conclusion

Spring Cloud Gateway simplifies the management of microservices interactions by providing a single entry point for client requests. Its rich set of features, built on the robust Spring ecosystem, makes it an excellent choice for building and managing API gateways.

By following the steps in this blog post, you've set up two simple microservices and used Spring Cloud Gateway to route requests to these services. Embrace the power of Spring Cloud Gateway and streamline your microservices architecture today!