KF7014 – Workshop week 03

API Gateway implementation

Introduction

In this worksheet, you will implement API Gateway using Spring Boot and Spring Cloud Gateway. You will build an order processing application that illustrates how an API Gateway can be used to invoke each service to retrieve customer and product data using the Customer and Product microservice, respectively.

Java-based frameworks

Spring Boot and Spring Cloud Gateway are both frameworks that are built on top of the **Spring** framework and are used for building Java-based applications. However, they serve different purposes and are typically used in different parts of an application.

> **Spring Boot** is a framework for building standalone, production-grade Spring-based applications. It provides a pre-configured set of libraries and components that can be easily integrated into new applications, and it can automatically configure and set up the application for you. This makes it a great tool for quickly developing and deploying new Spring-based applications.
>
> **Spring Cloud Gateway** is a framework for building microservices-based applications. It provides support for routing and filtering incoming requests to microservices, as well as support for other common features needed to build a microservices architecture. It is typically used as an API gateway in a microservices-based application.

Spring Cloud Gateway

Spring Cloud Gateway https://cloud.spring.io/spring-cloud-gateway/ is an API gateway framework built on top of several frameworks, including Spring Framework 5, Spring Boot 2, and Spring Webflux, which is a reactive web framework that is part of Spring Framework 5 and built on Project Reactor. Project Reactor is an NIO-based reactive framework for the JVM that provides the Mono abstraction.

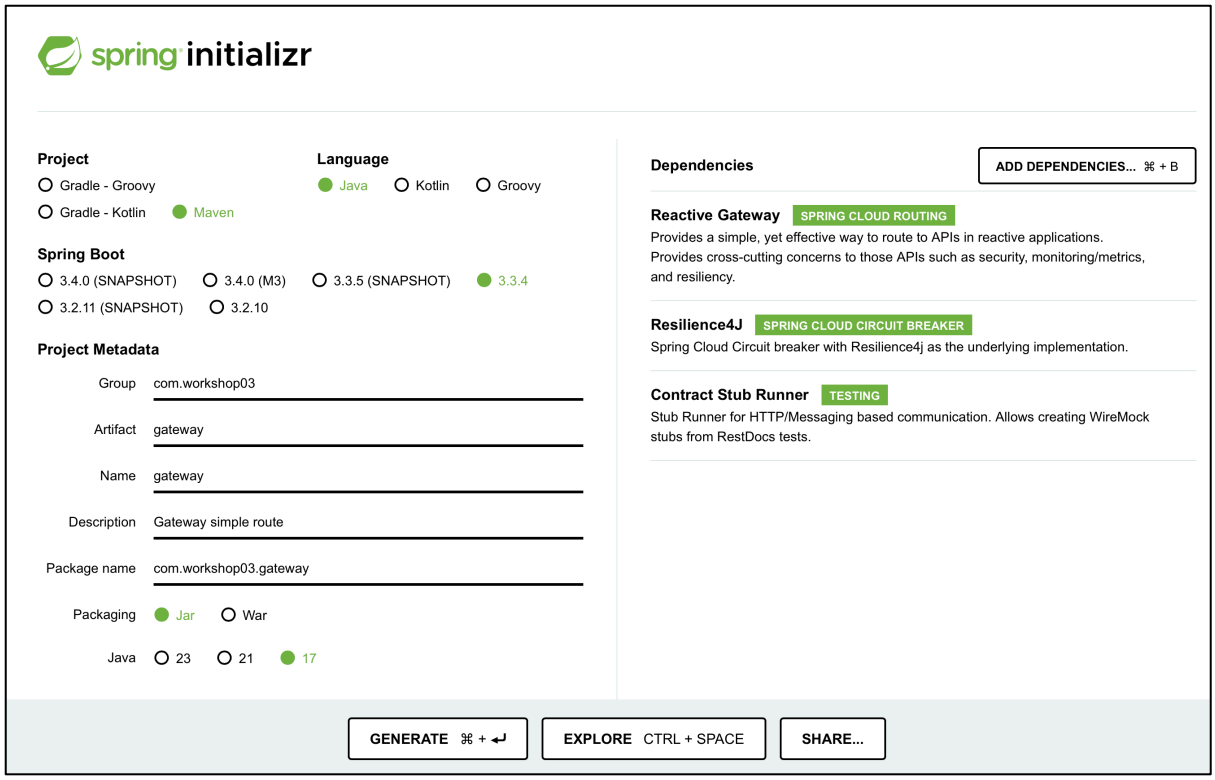
Spring Cloud Gateway provides a simple yet comprehensive way to do the following:

- Route requests to backend services.
- Implement request handlers that perform API composition.
- Handle edge functions such as authentication.

Exercise 1: Creating a simple route

Step 1: Create the project

- Create a new project with Spring Initializr https://start.spring.io
- Configure the project with the following:
    - Project: Maven
    - Language: Java
    - Dependencies: Reactive Gateway, Resilience4J, and Contract Stub Runner



- Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.
- Checking the `pom.xml` file, you should have the dependency included:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Step 2: Create a simple route

The Spring Cloud Gateway uses routes to process requests to downstream services. In this worksheet, we route all of our requests to HTTPBin.

- In `Application.java`, create a new `Bean` of type `RouteLocator`.

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes().build();
}
```

The `myRoutes` method takes in a `RouteLocatorBuilder` that can be used to create routes.

Exercise 2: Creating a route with predicates and filters

In addition to creating routes, RouteLocatorBuilder lets you add predicates and filters to your routes, so that you can route handle based on certain conditions as well as alter the request/response as you see fit.

- Create a route that routes a request to `https://httpbin.org/get` when a request is made to the Gateway at `/get`. In the configuration of this route, we add a **filter** that adds the `Hello` **request header** with a **value** of `World` to the request before it is routed.

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(p -> p
            .path("/get")
            .filters(f -> f.addRequestHeader("Hello", "World"))
            .uri("http://httpbin.org:80"))
        .build();
}
```

- To test the simple Gateway, we can run Application.java on port 8080. Once the application is running, make a request to http://localhost:8080/get.
- If PowerShell is installed on your device,, you can do so by using the following cURL command in your terminal:

```
$ curl http://localhost:8080/get
```

- If using Linux-based systems, you can use any tools equivalent to PowerShell.
- You should receive a response back that looks similar to the following output:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Forwarded": "proto=http;host=\"localhost:8080\";for=\"0:0:0:0:0:0:0:1:56207\"",
    "Hello": "World",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.54.0",
    "X-Forwarded-Host": "localhost:8080"
  },
  "origin": "0:0:0:0:0:0:0:1, 73.68.251.70",
  "url": "http://localhost:8080/get"
}
```

Exercise 3: Creating a route with CircuitBreaker

Since the services behind the Gateway could potentially behave poorly and affect the clients, you might want to wrap the routes in circuit breakers. You can do so in the **Spring Cloud Gateway** by using the **Resilience4J Spring Cloud CircuitBreaker** implementation. This is implemented through a simple filter that you can add to your requests.

Here we use HTTPBin's delay API, which waits a certain number of seconds before sending a response. Since this API could potentially take a long time to send its response, we can wrap the route that uses this API in a circuit breaker.

- Add a new route to the `RouteLocator` object:

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(p -> p
            .path("/get")
            .filters(f -> f.addRequestHeader("Hello", "World"))
            .uri("http://httpbin.org:80"))
        .route(p -> p
            .host("*.circuitbreaker.com")
            .filters(f -> f.circuitBreaker(config -> config.setName("mycmd")))
            .uri("http://httpbin.org:80")).
        build();
}
```

Note that here we used the host predicate (instead of the path predicate). This means that, as long as the host is `circuitbreaker.com`, the request is routed to `HTTPBin` and wrap that request in a circuit breaker. We do so by applying a filter to the route. We can configure the circuit breaker filter by using a configuration object. Here, the circuit breaker is named `mycmd`.

- To test this new route, start the application, but, this time, make a request to `/delay/3` and include a Host header that has a host of `circuitbreaker.com`. Otherwise, the request is not routed. Use the following cURL command:

```
$ curl --dump-header - --header 'Host: www.circuitbreaker.com' http://localhost:8080/delay/3
```

Here, we use `--dump-header` to see the response headers. The `-` after `--dump-header` tells `cURL` to print the headers to `stdout`.

After using this command, you should see the following in your terminal:

```
HTTP/1.1 504 Gateway Timeout
content-length: 0
```

The circuit breaker timed out while waiting for the response from `HTTPBin`.

When the circuit breaker times out, you can optionally provide a **fallback**, so that clients do not receive a 504 but something more meaningful.

- Modify the circuit breaker filter to provide a URL to call in the case of a timeout:

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder, UriConfiguration uriConfiguration) {
    String httpUri = uriConfiguration.getHttpbin();
    return builder.routes()
        .route(p -> p
            .path("/get")
            .filters(f -> f.addRequestHeader("header", "value"))
            .uri(httpUri))
        .route(p -> p
            .host("*.circuitbreaker.com")
            .filters(f -> f
                .circuitBreaker(config -> config
                    .setName("mycmd")
                    .setFallbackUri("forward:/fallback")))
            .uri(httpUri))
        .build();
}
```

Now, when the circuit breaker wrapped route times out, it calls `/fallback` in the Gateway application. Now we can add the `/fallback` endpoint to the application.

- In Application.java, add the `@RestController` class level annotation and then add the following `@RequestMapping` to the class:

```
@RequestMapping("/fallback")
public Mono<String> fallback() {
    return Mono.just("This is a fallback message for delays.");
}
```

- To test this new fallback functionality, restart the application and again issue the following cURL command:

```
$ curl --dump-header - --header 'Host: www.circuitbreaker.com' http://localhost:8080/delay/3
```

With the fallback in place, you see a `200` back from the Gateway with the response body of `fallback`.

```
HTTP/1.1 200 OK
transfer-encoding: chunked
Content-Type: text/plain;charset=UTF-8

fallback
```

If you receive error from subsequent running of the code, run this command to reset curl in PowerShell:

```
Remove-item alias:curl
```

Exercise 3: Testing the Gateway

As a good developer, you should write some tests to make sure the Gateway is doing what you expect it should.

Step 1: Set generic configuration

In most cases, we want to limit the dependencies on outside resources, especially in unit tests, so we should not depend on HTTPBin. One solution to this problem is to make the URI in the routes configurable, so we can change the URI if we need to.

- In `Application.java`, create a new class called `UriConfiguration`:

```
@ConfigurationProperties
class UriConfiguration {

    private String httpbin = "http://httpbin.org:80";

    public String getHttpbin() {
        return httpbin;
    }

    public void setHttpbin(String httpbin) {
        this.httpbin = httpbin;
    }
```

To enable `ConfigurationProperties`, you need to also add a class-level annotation to `Application.java`.

```
@EnableConfigurationProperties(UriConfiguration.class)
```

The complete `Application.java`:

GatewayApplication.java

```java
package com.workshop03.gateway;

import reactor.core.publisher.Mono;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.workshop03.gateway.GatewayApplication;
import com.workshop03.gateway.UriConfiguration;

@SpringBootApplication
@EnableConfigurationProperties(UriConfiguration.class)
@RestController
public class GatewayApplication {

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

    @Bean
    public RouteLocator myRoutes(RouteLocatorBuilder builder, UriConfiguration uriConfiguration) {
        String httpUri = uriConfiguration.getHttpbin();
        return builder.routes()
            .route(p -> p
                .path("/get")
                .filters(f -> f.addRequestHeader("header", "value"))
                .uri(httpUri))
            .route(p -> p
                .host("*.circuitbreaker.com")
                .filters(f -> f
                    .circuitBreaker(config -> config
                        .setName("mycmd")
                        .setFallbackUri("forward:/fallback")))
                .uri(httpUri))
            .build();
    }

    @RequestMapping("/fallback")
    public Mono<String> fallback() {
        return Mono.just("This is a fallback message for delays.");
    }
}

@ConfigurationProperties
class UriConfiguration {

    private String httpbin = "http://httpbin.org:80";

    public String getHttpbin() {
        return httpbin;
    }

    public void setHttpbin(String httpbin) {
        this.httpbin = httpbin;
    }
}
```

Step 2: Writing tests

- Create a new class called `ApplicationTest` and add the following code:

GatewayApplicationTests.java

```java
package com.workshop03.gateway;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.cloud.contract.wiremock.AutoConfigureWireMock;
import org.springframework.test.web.reactive.server.WebTestClient;

import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static org.assertj.core.api.Assertions.*;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
properties = {"httpbin=http://localhost:${wiremock.server.port}"})
@AutoConfigureWireMock(port = 0)
class GatewayApplicationTests {

    @Autowired
    private WebTestClient webClient;

    @Test
    public void contextLoads() throws Exception {
        //Stubs
        stubFor(get(urlEqualTo("/get"))
                .willReturn(aResponse()
                    .withBody("{\"headers\":{\"Hello\":\"World\"}}")
                    .withHeader("Content-Type", "application/json")));
        stubFor(get(urlEqualTo("/delay/3"))
            .willReturn(aResponse()
                .withBody("no fallback")
                .withFixedDelay(3000)));

        webClient
            .get().uri("/get")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.headers.header").isEqualTo("value");

        webClient
            .get().uri("/delay/3")
            .header("Host", "www.circuitbreaker.com")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .consumeWith(
                response -> assertThat(response.getResponseBody()).isEqualTo("fallback".getBytes()));
    }
}
```

The test takes advantage of **WireMock** from Spring Cloud Contract stand up a server that can mock the APIs from `HTTPBin`. The first thing to notice is the use of `@AutoConfigureWireMock(port = 0)`. This annotation starts WireMock on a random port for us.

Next, note that we take advantage of the `UriConfiguration` class and set the `httpbin` property in the `@SpringBootTest` annotation to the WireMock server running locally. Within the test, we then setup “stubs” for the HTTPBin APIs we call through the Gateway and mock the expected behaviour. Finally, `WebTestClient` is used to make requests to the Gateway and validate the responses.

Running `ApplicationTest`, you will receive an output similar to:

```
Matched response definition:
{
  "status" : 200,
  "body" : "{\"headers\":{\"Hello\":\"World\"}}",
  "headers" : {
    "Content-Type" : "application/json"
  }
}

Response:
HTTP/1.1 200
Content-Type: [application/json]
Matched-Stub-Id: [3af7493c-b562-4936-b93d-4bc9ce430cda]

2024-10-15T02:41:29.680+01:00  INFO 21507 --- [qtp939871489-36] WireMock                                 : Request received:
127.0.0.1 - GET /delay/3

Accept-Encoding: [gzip]
User-Agent: [ReactorNetty/1.1.19]
Accept: [*/*]
WebTestClient-Request-Id: [2]
Forwarded: [proto=http;host=www.circuitbreaker.com;for="127.0.0.1:51037"]
X-Forwarded-For: [127.0.0.1]
X-Forwarded-Proto: [http]
X-Forwarded-Port: [80]
X-Forwarded-Host: [www.circuitbreaker.com]
Host: [localhost:11091]
Content-Length: [0]



Matched response definition:
{
  "status" : 200,
  "body" : "no fallback",
  "fixedDelayMilliseconds" : 3000
}

Response:
HTTP/1.1 200
Matched-Stub-Id: [0395a9ef-a178-4779-88ba-74ad0114b5da]
```

Now you have created the first Spring Cloud Gateway application! …

Directed learning: Creating a route for a microservice

Use the microservice developed in workshop 02, and develop a gateway application for this microservice.