Spring Boot Microservices: Spring Security with Api Gateway Implementation

DECEMBRE 2024

Article written by **Hernadez Ebolo Ekangwo,** a cybersecurity analyst specializing in incident response and threat intelligence, at RHOPEN Labs.

Disclaimer of Liability

The information contained in this article is intended for educational and awareness purposes only, especially for cybersecurity professionals in francophone Africa.

The examples and scenarios described here should not be used for illegal or unethical activities. The authors and distributors of this article are not liable for any misuse or malevolence of the information provided.

Businesses and individuals should consult with cybersecurity professionals for specific advice tailored to their unique needs and contexts. Any attempt to exploit the security breaches described in this article for personal or business gain is strongly condemned and may result in legal action

In this article we shall be discussing about implementing an api gateway for our microservices in the same service as spring security.

In my past years as a senior software engineer, I have noticed and particularly worked on both architectural type Monolith and Microservices. From observations Microservice architecture has drastically increase it usage.

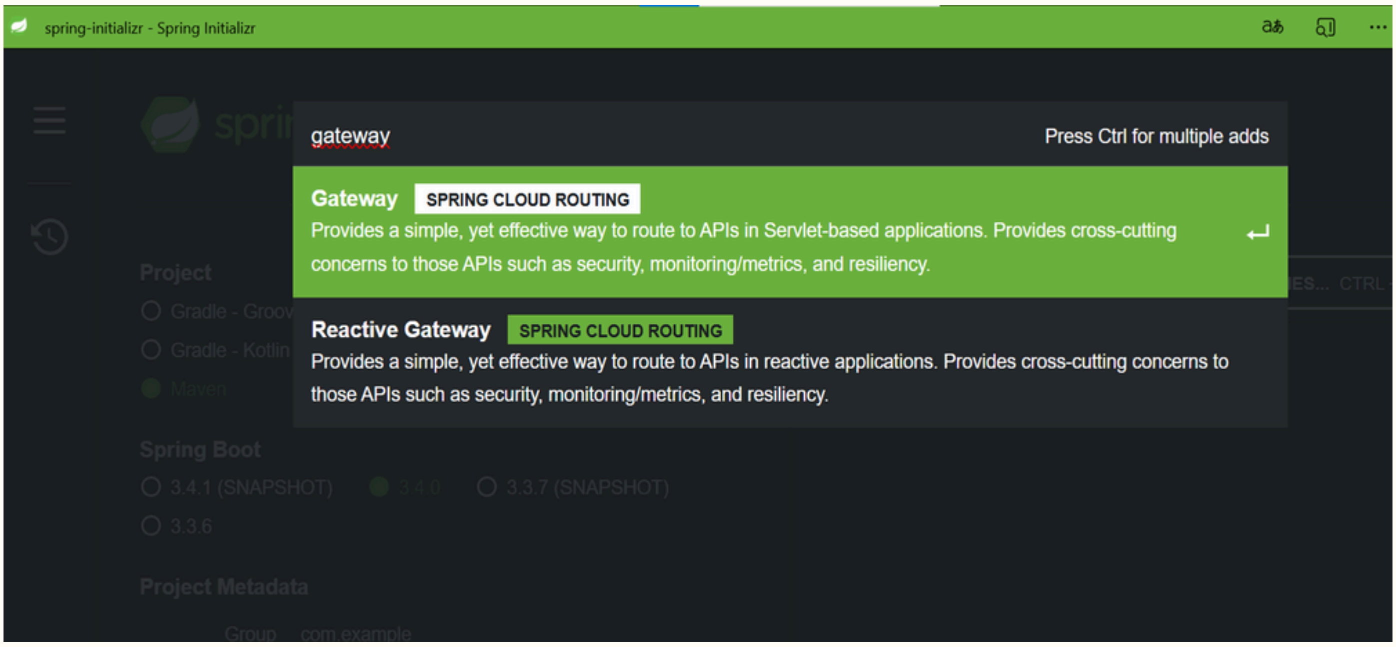
So, in this article we shall place our main focus on implementing an API gateway for our spring boot microservice architecture directly into our spring security microservice. For those not verse with spring security an article will be published related to that.

What exactly is spring security: Spring Security is a framework that provides authentication, authorization, and protection against common attacks. With first class support for securing both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.
With this perspective instead of creating a separate microservice for our API gateway we decided to make it up in the spring security for the following reasons:

- Centralized security model
- Performance and Latency improvement
- Enhanced Scalability and Flexibility
- Improve developer productivity

For those who have implemented API gateway precedent to December 2023 must have noticed there was only a single module for Spring cloud gateway which was meant for Reactive Solution (Webflux) but not MVC so later than December 2023 a new Spring cloud gateway was introduce called Spring Cloud Gateway Server MVC built on Spring Boot and Spring WebMvc.fn. which actually unblock the complexity of implementing an api gateway for your MVC spring boot application.
Base on this the implementation is somehow different from the original Spring Cloud Gateway something which was some how difficult to get the right annotations from the beginning. To demonstrate this I have created two spring services one which is the spring security and the other normal user services.

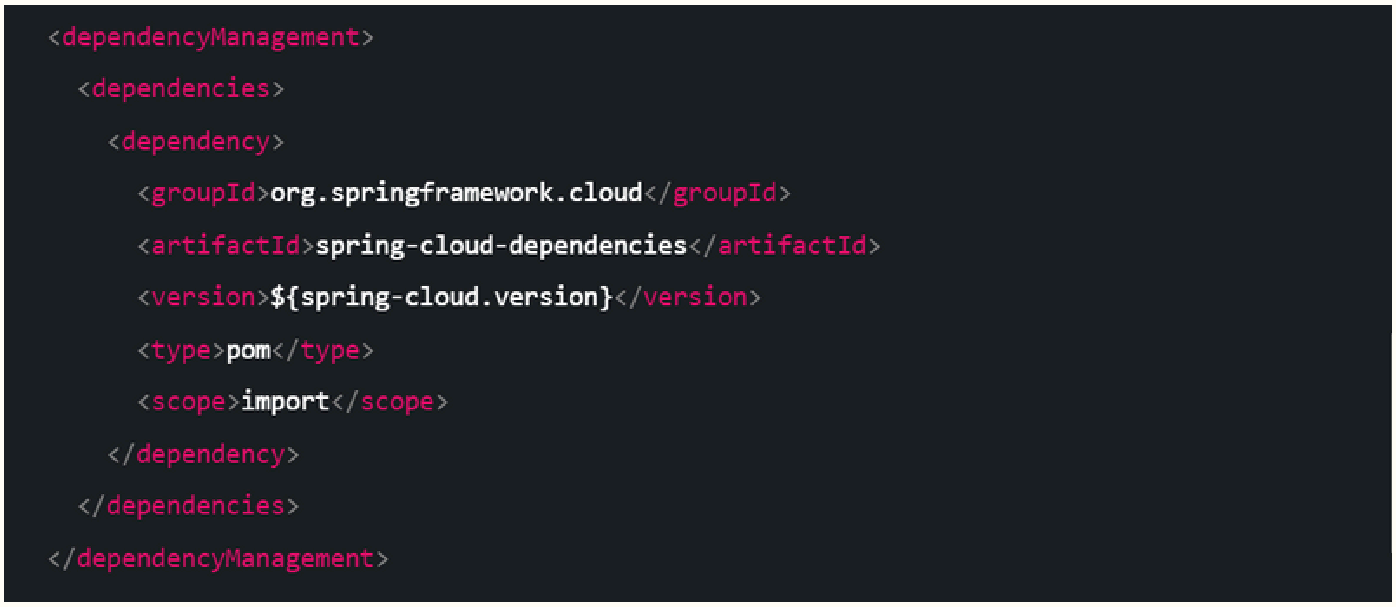
Let’s goooooooooo
First thing first, in the spring pom.xml file the spring gate was included we use spring-initializr for that. take note like I said earlier there are two and the right one is the Gateway and not Reactive Gateway if you are architecture is an MVC.



Gateway SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs in Servlet-based applications. Provides cross-cutting concerns to those APIs such as security, monitoring/metrics, and resiliency.

Reactive Gateway SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs in reactive applications. Provides cross-cutting concerns to those APIs such as security, monitoring/metrics, and resiliency.

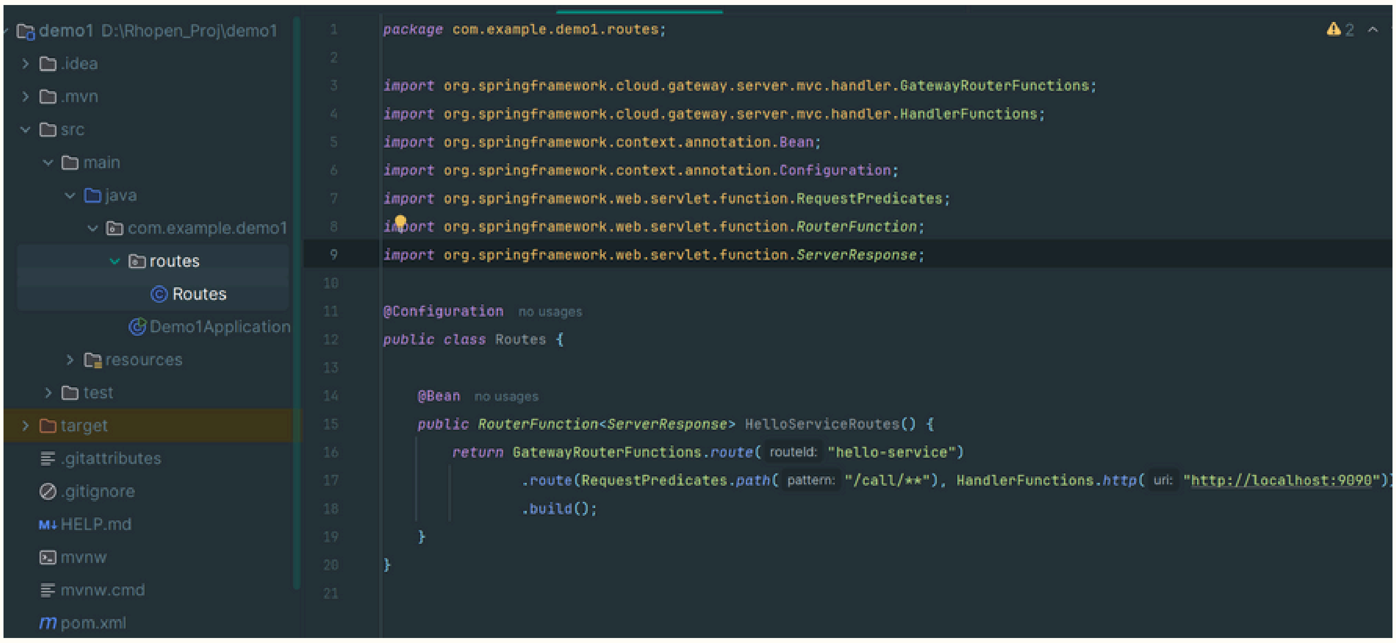
```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway-mvc</artifactId>
</dependency>
```



```
<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>
```

Those are the dependency to be included for the gateway. After this we are sure and certain our spring security service now includes gateway properties.
Next, we have two methods of implementing our routing for the gateway (using Java class and using Property file) we shall have an overview of each.

Approach 1: Using Java Class-: This approach relies on creating a package name route in the root project then creating a class name as Route.class with the following route config for example.



```java
package com.example.demo1.routes;

import org.springframework.cloud.gateway.server.mvc.handler.GatewayRouterFunctions;
import org.springframework.cloud.gateway.server.mvc.handler.HandlerFunctions;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.function.RequestPredicates;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

@Configuration  no usages
public class Routes {

    @Bean  no usages
    public RouterFunction<ServerResponse> HelloServiceRoutes() {
        return GatewayRouterFunctions.route( routeId: "hello-service")
                .route(RequestPredicates.path( pattern: "/call/**"), HandlerFunctions.http( uri: "http://localhost:9090"))
                .build();
    }
}
```

As we can see the route config.. I use the RequestPredicates to define the path which I what to contact for the other service and the HandlerFunction to indicate the Url server of the service.
This alone is sufficient for our gateway but remember that this is included with spring security and some config has to be done effectively.
Firstly we need to resolve CORS issue with our spring security so the frontend can asses all endpoints in the service both on the spring security service and the second microservice.

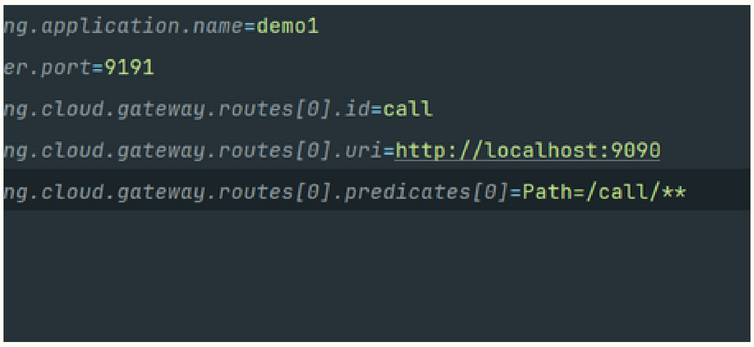
So our security microservice will loke like this

```
package com.security.rhopenlabs.gateway.config;import org.springframework.context.annotation.Bean;import
org.springframework.context.annotation.Configuration;import org.springframework.security.config.Customizer;import
org.springframework.security.config.annotation.web.builders.HttpSecurity;import
org.springframework.security.web.SecurityFilterChain;import org.springframework.web.cors.CorsConfiguration;import
org.springframework.web.cors.CorsConfigurationSource;import
org.springframework.web.cors.UrlBasedCorsConfigurationSource;import java.util.Arrays;import
java.util.List;@Configurationpublic class SecurityConfig {    private final String[] freeResourceUrls = {"/swagger-
ui.html", "/swagger-ui/**", "/v3/api-docs/**", "/swagger-resources/**", "/aggregate/**","/call/**"};    @Bean    public
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {        return
httpSecurity.authorizeHttpRequests(authorize -> authorize
.requestMatchers(freeResourceUrls).permitAll()                        .anyRequest().authenticated())
.cors(corsConfigurer -> corsConfigurer.configurationSource(corsConfigurationSource()))
.oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))                .build();    }    @Bean
CorsConfigurationSource corsConfigurationSource() {        CorsConfiguration configuration = new
CorsConfiguration();        configuration.setAllowedOrigins(List.of("*"));
configuration.setAllowedMethods(Arrays.asList("GET","POST"));        configuration.setAllowedHeaders(List.of("*"));
UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", configuration);        return source;    }}
```

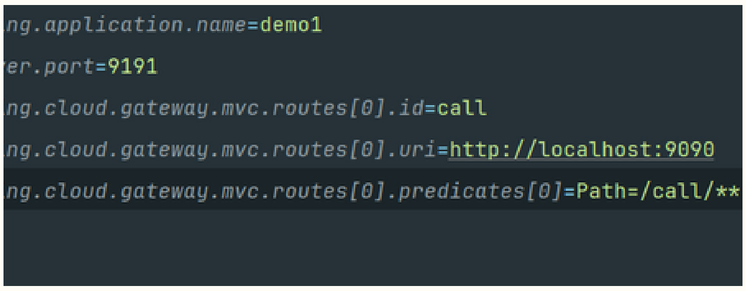
once this is done we can now acces the other service with juste the gateway url and specify the endpoint we want to get in the other service.

Approach 2: Property file (application.properties) this appraoche include defining the config in the application.properties file remember I said this config is different from the first implementation of Reactive application spring cloud gateway..

The difficulties were to get the right config:



```
ng.application.name=demo1
er.port=9191
ng.cloud.gateway.routes[0].id=call
ng.cloud.gateway.routes[0].uri=http://localhost:9090
ng.cloud.gateway.routes[0].predicates[0]=Path=/call/**
```



```
ng.application.name=demo1
er.port=9191
ng.cloud.gateway.mvc.routes[0].id=call
ng.cloud.gateway.mvc.routes[0].uri=http://localhost:9090
ng.cloud.gateway.mvc.routes[0].predicates[0]=Path=/call/**
```

Actually, this config was for the first spring cloud gate which meant for Reactive applications
After many tries and documentation, I found out for Spring cloud gateway mvc is different something no other blocs never mentioned before since they make use of java class config.

With this you're all set for you API gateway remember to always apply CORS config in your entire environment. See you shortly

ABOUT THE REACTOR IN THE ARTICLE:

HERNADEZ EBOLO EKANGWO IS A CYBERSECURITY ANALYST SPECIALIZING IN INCIDENT RESPONSE AND THREAT INTELLIGENCE. HE ALSO HAS EXPERTISE IN CYBERSECURITY ENGINEERING.

ACTIVE IN THE FIELD OF CYBERSECURITY, HERNADEZ WORKS AS A CYBER SECURITY ANALYST, SPECIALIZING IN INCIDENT RESPONSE AND THREAT INTELLIGENCE. UNDER THE DIRECTION OF **FRANÇOIS-XAVIER DJIMGOU NGAMENI,** PRESIDENT OF RHOPEN LABS AND CYBERSECURITY EXPERT.

Lien LinkedIn

About RHOPEN Labs

Subsidiary of the French technology company RHOPEN, RHOPEN Labs is a Cameroonian company with a capital of 20,000,000 CFA francs, innovative company in the field of Cybersecurity and Cloud/DevOps and dedicated to protecting African organizations against digital threats. With a team of experts and leading-edge endogenous solutions, RHOPEN Labs is committed to providing world-class security services.