**API Gateway with Spring Boot Security: OAuth2, Google Developer API, Rate Limiting, and Routing**

**1. Introduction to API Gateway**

An **API Gateway** acts as an entry point for clients, managing:

* **Routing**: Forwarding requests to appropriate microservices.
* **Authentication and Authorization**: Ensuring secure access using OAuth2.
* **Rate Limiting**: Preventing abuse by controlling traffic.

We will use **Spring Cloud Gateway** for API Gateway functionality.

**2. Project Setup**

**Dependencies**

Add the following dependencies to your pom.xml:

<dependencies>

<!-- Spring Cloud Gateway -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<!-- Spring Security -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

<!-- Redis for Rate Limiting -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis-reactive</artifactId>

</dependency>

<!-- Actuator for Monitoring -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

**Application Properties**

Add configurations in application.yml:

server:

port: 8080

spring:

security:

oauth2:

client:

registration:

google:

client-id: YOUR\_GOOGLE\_CLIENT\_ID

client-secret: YOUR\_GOOGLE\_CLIENT\_SECRET

scope: openid, email, profile

redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"

client-authentication-method: post

authorization-grant-type: authorization\_code

provider:

google:

authorization-uri: https://accounts.google.com/o/oauth2/v2/auth

token-uri: https://oauth2.googleapis.com/token

user-info-uri: https://openidconnect.googleapis.com/v1/userinfo

gateway:

routes:

- id: service1

uri: http://localhost:8081

predicates:

- Path=/service1/\*\*

filters:

- RewritePath=/service1/(?<segment>.\*), /${segment}

- id: service2

uri: http://localhost:8082

predicates:

- Path=/service2/\*\*

- Method=GET

filters:

- RewritePath=/service2/(?<segment>.\*), /${segment}

**3. Implementing Features**

**OAuth2 Authentication with Google**

1. **Google Developer Console Setup**:
   * Go to the Google Cloud Console.
   * Create a project and enable the **Google+ API** or **Google OAuth API**.
   * Generate OAuth2 credentials (Client ID and Client Secret).
   * Set http://localhost:8080/login/oauth2/code/google as the redirect URI.
2. **OAuth2 Security Configuration**: Create a SecurityConfig class for OAuth2 login:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.oauth2.client.web.OAuth2LoginAuthenticationFilter;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.oauth2Login(); // Enable OAuth2 login

return http.build();

}

}

1. **Securing Routes**: Only authenticated users can access the gateway routes:
   * Add .authorizeRequests().antMatchers("/service1/\*\*").authenticated() in the SecurityConfig.

**Rate Limiting**

1. **Redis Configuration**: Install Redis for reactive rate limiting.
   * Use Docker for local setup:

docker run -d --name redis -p 6379:6379 redis

1. **Gateway Rate Limiting Filter**: Configure rate limiting using Redis:

spring:

cloud:

gateway:

routes:

- id: rate-limit-service

uri: http://localhost:8083

predicates:

- Path=/ratelimited/\*\*

filters:

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 10

redis-rate-limiter.burstCapacity: 20

1. **Enable Rate Limiter in Gateway**:
   * Implement a RedisRateLimiter bean:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.cloud.gateway.filter.ratelimit.RedisRateLimiter;

@Configuration

public class RateLimiterConfig {

@Bean

public RedisRateLimiter redisRateLimiter() {

return new RedisRateLimiter(10, 20); // Replenish rate and burst capacity

}

}

**Routing**

1. **Routing Configuration**: Use **Spring Cloud Gateway** for routing:
   * Requests to /service1/\*\* will be forwarded to http://localhost:8081.
   * Requests to /service2/\*\* will be forwarded to http://localhost:8082.
2. **Custom Predicates and Filters**: Add custom predicates like HTTP method, headers, etc.

Example for a custom filter:

import org.springframework.cloud.gateway.filter.GatewayFilter;

import org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;

import org.springframework.stereotype.Component;

@Component

public class CustomHeaderFilter extends AbstractGatewayFilterFactory<Object> {

@Override

public GatewayFilter apply(Object config) {

return (exchange, chain) -> {

exchange.getRequest()

.mutate()

.header("X-Custom-Header", "CustomValue")

.build();

return chain.filter(exchange);

};

}

}

1. **Global Filters**: Apply global filters to log requests or transform headers for all routes:

import org.springframework.cloud.gateway.filter.GlobalFilter;

import org.springframework.stereotype.Component;

import reactor.core.publisher.Mono;

@Component

public class LoggingFilter implements GlobalFilter {

@Override

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

System.out.println("Request Path: " + exchange.getRequest().getPath());

return chain.filter(exchange);

}

}

**4. Testing the API Gateway**

1. **Authentication**:
   * Access http://localhost:8080/service1/\*\*.
   * You will be redirected to Google's OAuth2 login page.
2. **Rate Limiting**:
   * Send multiple requests to /ratelimited/\*\*.
   * After exceeding the limit, you will receive a 429 Too Many Requests error.
3. **Routing**:
   * Send requests to /service1/\*\* or /service2/\*\* to verify routing.

**5. Monitoring with Actuator**

Enable **Spring Boot Actuator** for monitoring API Gateway:

management:

endpoints:

web:

exposure:

include: "\*"

endpoint:

health:

show-details: always

**Summary**

This API Gateway setup includes:

* **OAuth2 Security**: Using Google Developer API for authentication.
* **Rate Limiting**: Protecting APIs from abuse with Redis-based rate limiting.
* **Routing**: Forwarding requests to backend microservices based on routes.
* **Custom Filters**: Adding custom logic for headers, logging, or transformations.