Secure Rate Limiting with Spring Cloud Gateway

Microservices, Performance, Security, Spring Boot, Spring Cloud

Secure Rate Limiting with Spring Cloud Gateway

By piotr.minkowski • May 21, 2021 • 6

```
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange(exchanges -> exchanges.anyExchange().authenticated())
            .oauth2Login(withDefaults());
        http.csrf().disable();
        return http.build();
    }
```

In this article, you will learn how to enable rate limiting for an authenticated user with Spring Cloud Gateway. Why it is important? API gateway is an entry point to your microservices system. Therefore, you should provide there a right level of security. Rate limiting can prevent your API against DoS attacks and limit web scraping.

You can easily configure rate limiting with Spring Cloud Gateway. For a basic introduction to this feature, you may refer to my article **Rate Limiting in Spring Cloud Gateway with Redis**. Similarly, today we will also use Redis as a backend for a rate limiter. Moreover, we will configure an HTTP basic authentication. Of course, you can provide some more advanced authentication mechanisms like an X509 certificate or OAuth2 login. If you think about it, read my article **Spring Cloud Gateway OAuth2 with Keycloak**.

Source Code

If you would like to try it by yourself, you may always take a look at my source code. In order to do that you need to clone my repository **sample-spring-cloud-gateway**. Then you should go to the `src/test/java` directory, and just follow my instructions in the next sections.

1. Dependencies

Let's start with dependencies. Since we will create an integration test, we need some additional libraries. Firstly, we will use the Testcontainers library. It allows us to run Docker containers during the JUnit test. We will use it for running Redis and a mock server, which is responsible for mocking a downstream service. Of course, we need to include a starter with Spring Cloud Gateway and Spring Data Redis. To implement an HTTP basic authentication we also need to include Spring Security. Here's a full list of required dependencies in Maven `pom.xml`.

```xml
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.testcontainers</groupId>
   <artifactId>mockserver</artifactId>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mock-server</groupId>
    <artifactId>mockserver-client-java</artifactId>
    <scope>test</scope>
</dependency>
```

2. Configure an HTTP Basic Authentication

In order to configure an HTTP basic authentication, we need to create the `@Configuration` bean annotated with `@EnableWebFluxSecurity`. That's because Spring Cloud Gateway is built on top of Spring WebFlux and Netty. Also, we will create a set of test users with `MapReactiveUserDetailsService`.

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

   @Bean
   public SecurityWebFilterChain filterChain(ServerHttpSecurity http) {
      http.authorizeExchange(exchanges ->
         exchanges.anyExchange().authenticated())
            .httpBasic();
      http.csrf().disable();
      return http.build();
   }

   @Bean
   public MapReactiveUserDetailsService users() {
      UserDetails user1 = User.builder()
            .username("user1")
            .password("{noop}1234")
            .roles("USER")
            .build();
      UserDetails user2 = User.builder()
            .username("user2")
            .password("{noop}1234")
            .roles("USER")
            .build();
      UserDetails user3 = User.builder()
            .username("user3")
            .password("{noop}1234")
            .roles("USER")
```

```
                .build();
        return new MapReactiveUserDetailsService(user1, user2, user3);
    }
}
```

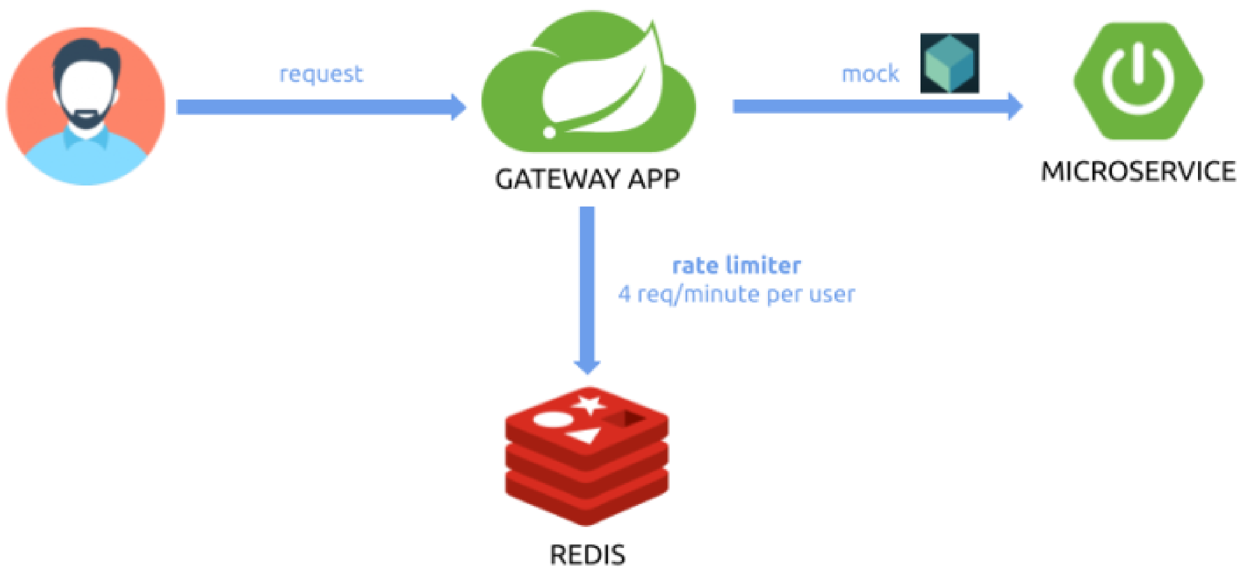
3. Configure Spring Cloud Gateway Rate Limiter key

A request rate limiter feature needs to be enabled using the component called `GatewayFilter`. This filter takes an optional `keyResolver` parameter. The `KeyResolver` interface allows you to create pluggable strategies derive the key for limiting requests. In our case, it will be a user login. Once a user has been successfully authenticated, its login is stored in the Spring `SecurityContext`. In order to retrieve the context for a reactive application, we should use `ReactiveSecurityContextHolder`.

```
@Bean
KeyResolver authUserKeyResolver() {
    return exchange -> ReactiveSecurityContextHolder.getContext()
            .map(ctx -> ctx.getAuthentication()
               .getPrincipal().toString());
}
```

4. Test Scenario

In the test scenario, we are going to simulate incoming traffic. Every single request needs to have a `Authorization` header with the user credentials. A single user may send 4 requests per minute. After exceeding that limit Spring Cloud Gateway will return the HTTP code `HTTP 429 - Too Many Requests`. The traffic is addressed to the downstream service. Therefore, we are running a mock server using Testcontainers.



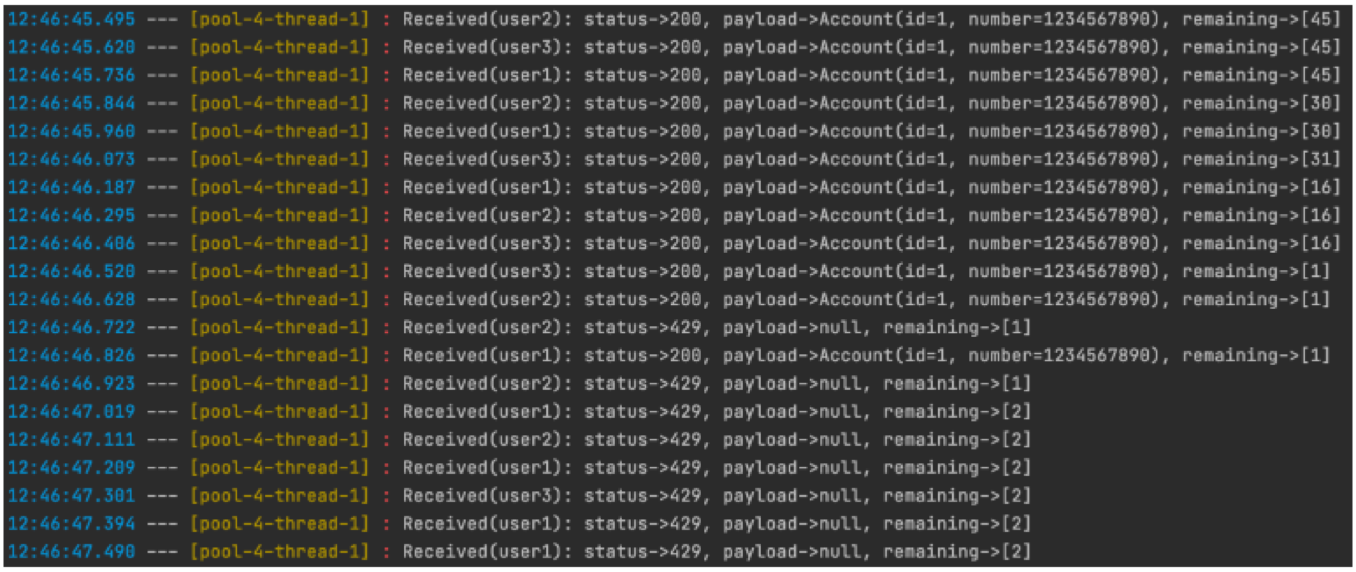
5. Testing Spring Cloud Gateway secure rate limiter

Finally, we may proceed to the test implementation. I will use JUnit4 since I used it before for the other examples in the sample repository. We have three parameters used for rate limiter configuration: `replenishRate`, `burstCapacity` and `requestedTokens`. Since we also allow less than 1 request per second we need to set the right values for `burstCapacity` and `requestedTokens`. In short, the `requestedTokens` property sets how many tokens a request costs. On the other hand, the `burstCapacity` property is the maximum number of requests (or cost) that is allowed for a user.

During the test we randomly set the username between `user1`, `user2` and `user3`. The test is repeated 20 times.

```
@SpringBootTest(webEnvironment =
   SpringBootTest.WebEnvironment.DEFINED_PORT,
                properties = {"rateLimiter.secure=true"})
@RunWith(SpringRunner.class)
public class GatewaySecureRateLimiterTest {

   private static final Logger LOGGER =
      LoggerFactory.getLogger(GatewaySecureRateLimiterTest.class);
   private Random random = new Random();

   @Rule
   public TestRule benchmarkRun = new BenchmarkRule();

   @ClassRule
   public static MockServerContainer mockServer =
      new MockServerContainer();
   @ClassRule
   public static GenericContainer redis =
      new GenericContainer("redis:5.0.6").withExposedPorts(6379);

   @Autowired
   TestRestTemplate template;

   @BeforeClass
   public static void init() {
      System.setProperty("spring.cloud.gateway.routes[0].id", "account-service");
      System.setProperty("spring.cloud.gateway.routes[0].uri", "http://" + mockServer.getHo
      System.setProperty("spring.cloud.gateway.routes[0].predicates[0]", "Path=/account/**"
      System.setProperty("spring.cloud.gateway.routes[0].filters[0]", "RewritePath=/account
      System.setProperty("spring.cloud.gateway.routes[0].filters[1].name", "RequestRateLimi
      System.setProperty("spring.cloud.gateway.routes[0].filters[1].args.redis-rate-limiter
      System.setProperty("spring.cloud.gateway.routes[0].filters[1].args.redis-rate-limiter
      System.setProperty("spring.cloud.gateway.routes[0].filters[1].args.redis-rate-limiter
      System.setProperty("spring.redis.host", redis.getHost());
      System.setProperty("spring.redis.port", "" + redis.getMappedPort(6379));
      new MockServerClient(mockServer.getContainerIpAddress(), mockServer.getServerPort())
            .when(HttpRequest.request()
                    .withPath("/1"))
            .respond(response()
                    .withBody("{\"id\":1,\"number\":\"1234567890\"}")
                    .withHeader("Content-Type", "application/json"));
   }

   @Test
   @BenchmarkOptions(warmupRounds = 0, concurrency = 1, benchmarkRounds = 20)
   public void testAccountService() {
      String username = "user" + (random.nextInt(3) + 1);
      HttpHeaders headers = createHttpHeaders(username,"1234");
      HttpEntity<String> entity = new HttpEntity<String>(headers);
      ResponseEntity<Account> r = template
         .exchange("/account/{id}", HttpMethod.GET, entity, Account.class, 1);
      LOGGER.info("Received({}): status->{}, payload->{}, remaining->{}",
            username, r.getStatusCodeValue(), r.getBody(), r.getHeaders().get("X-RateLimit-
    }

   private HttpHeaders createHttpHeaders(String user, String password) {
      String notEncoded = user + ":" + password;
      String encodedAuth = Base64.getEncoder().encodeToString(notEncoded.getBytes());
      HttpHeaders headers = new HttpHeaders();
      headers.setContentType(MediaType.APPLICATION_JSON);
      headers.add("Authorization", "Basic " + encodedAuth);
      return headers;
   }

}
```

Let’s run the test. Thanks to the `junit-benchmarks` library we may configure the number of rounds for the test. Each time I’m logging the response from the gateway that includes username, HTTP status, payload, and a header `X-RateLimit-Remaining` that shows a number of remaining tokens. Here’s the result.


```
12:46:45.495 --- [pool-4-thread-1] : Received(user2): status->200, payload->Account(id=1, number=1234567890), remaining->[45]
12:46:45.620 --- [pool-4-thread-1] : Received(user3): status->200, payload->Account(id=1, number=1234567890), remaining->[45]
12:46:45.736 --- [pool-4-thread-1] : Received(user1): status->200, payload->Account(id=1, number=1234567890), remaining->[45]
12:46:45.844 --- [pool-4-thread-1] : Received(user2): status->200, payload->Account(id=1, number=1234567890), remaining->[30]
12:46:45.960 --- [pool-4-thread-1] : Received(user1): status->200, payload->Account(id=1, number=1234567890), remaining->[30]
12:46:46.073 --- [pool-4-thread-1] : Received(user3): status->200, payload->Account(id=1, number=1234567890), remaining->[31]
12:46:46.187 --- [pool-4-thread-1] : Received(user1): status->200, payload->Account(id=1, number=1234567890), remaining->[16]
12:46:46.295 --- [pool-4-thread-1] : Received(user2): status->200, payload->Account(id=1, number=1234567890), remaining->[16]
12:46:46.406 --- [pool-4-thread-1] : Received(user3): status->200, payload->Account(id=1, number=1234567890), remaining->[16]
12:46:46.520 --- [pool-4-thread-1] : Received(user3): status->200, payload->Account(id=1, number=1234567890), remaining->[1]
12:46:46.628 --- [pool-4-thread-1] : Received(user2): status->200, payload->Account(id=1, number=1234567890), remaining->[1]
12:46:46.722 --- [pool-4-thread-1] : Received(user2): status->429, payload->null, remaining->[1]
12:46:46.826 --- [pool-4-thread-1] : Received(user1): status->200, payload->Account(id=1, number=1234567890), remaining->[1]
12:46:46.923 --- [pool-4-thread-1] : Received(user2): status->429, payload->null, remaining->[1]
12:46:47.019 --- [pool-4-thread-1] : Received(user1): status->429, payload->null, remaining->[2]
12:46:47.111 --- [pool-4-thread-1] : Received(user2): status->429, payload->null, remaining->[2]
12:46:47.209 --- [pool-4-thread-1] : Received(user1): status->429, payload->null, remaining->[2]
12:46:47.301 --- [pool-4-thread-1] : Received(user3): status->429, payload->null, remaining->[2]
12:46:47.394 --- [pool-4-thread-1] : Received(user1): status->429, payload->null, remaining->[2]
12:46:47.490 --- [pool-4-thread-1] : Received(user1): status->429, payload->null, remaining->[2]
```

Like this:

basic auth | rate limiting | Redis | Spring Boot | Spring Cloud | Spring Cloud Gateway | Spring Data Redis | Spring Security | testcontainers

Written by:

piotr.minkowski

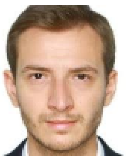
View All Posts →

6 COMMENTS

João Paulo August 9, 2021 4:08 pm

Hello Piotr! Thanks for sharing your knowledge! It was not clear to me how do I set requests per minute or requests per hour. Is it possible?

REPLY



piotr.minkowski August 31, 2021 3:32 pm

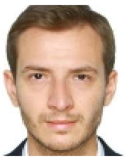
Yes, in that case, you have four requests per user per one minute.

REPLY

Tolu October 2, 2021 10:11 pm

Hi. Thank you for your tutorial. I have a few questions. Can I go ahead and ask? OR just drop you my email address and we can talk more via email?

REPLY



piotr.minkowski October 12, 2021 3:03 pm

Hi!
You can contact me at piotr.minkowski@gmail.com

REPLY

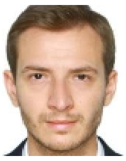
aylin January 19, 2022 2:32 pm

I have configured an endpoint, so that clients can call only 5 times per hour, then it should throw too many request exception. But after 15 minutes that i got too many request, i could do a successful call again.

```
redis-rate-limiter:
replenishRate: 1
burstCapacity: 3600
requestedTokens: 720
```

how should i update these values, so that only 5 request per hour would allowed?

REPLY



piotr.minkowski January 20, 2022 2:10 pm

Yes, it should work. Did you try it?

REPLY