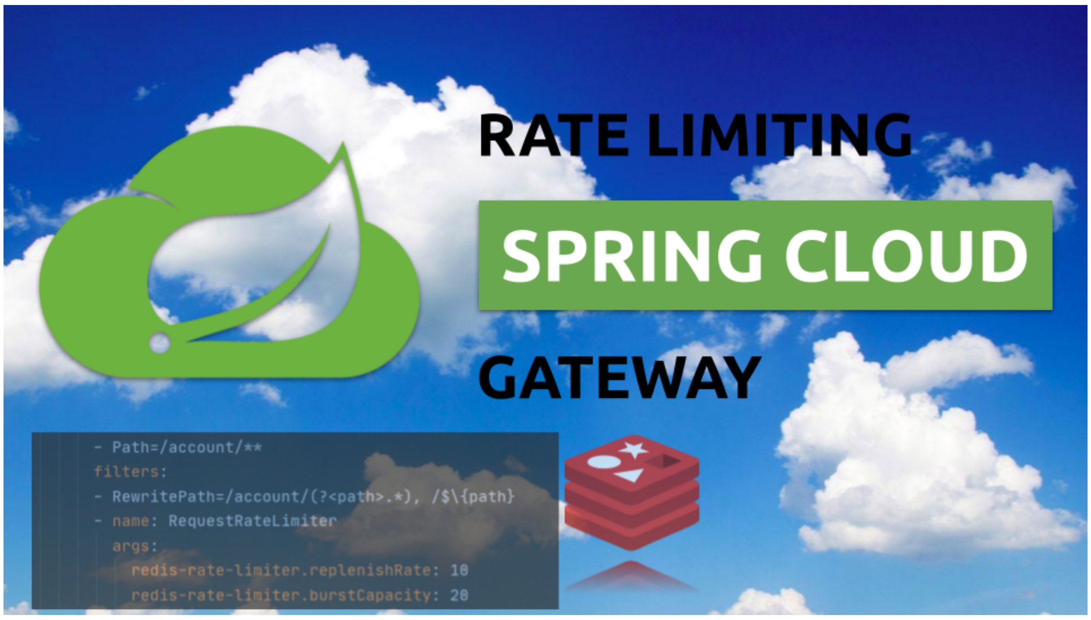
Piotr's TechBlog

Java, Spring, Kotlin, microservices, Kubernetes, containers

Performance, Spring Cloud

Rate Limiting In Spring Cloud Gateway With Redis

By piotr.minkowski • November 15, 2019 • 21



```
- Path=/account/**
filters:
- RewritePath=/account/(?<path>.*), /$\{path}
- name: RequestRateLimiter
  args:
    redis-rate-limiter.replenishRate: 10
    redis-rate-limiter.burstCapacity: 20
```

Currently **Spring Cloud Gateway** is second the most popular Spring Cloud project just after Spring Cloud Netflix (in terms of a number of stars on GitHub). It has been created as a successor of Zuul proxy in the Spring Cloud family. This project provides an API Gateway for microservices architecture, and is built on top of reactive **Netty** and Project **Reactor**. It is designed to provide a simple, but effective way to route to APIs and address such popular concerns as security, monitoring/metrics, and resiliency.
Spring Cloud Gateway offers you many features and configuration options. Today I'm going to focus on the single one, but very interesting aspect of gateway configuration – **rate limiting**. A rate limiter may be defined as a way to control the rate of traffic sent or received on the network. We can also define a few types of rate limiting. Spring Cloud Gateway currently provides a **Request Rate Limiter**, which is responsible for restricting each user to N requests per second. When using `RequestRateLimiter` with Spring Cloud Gateway we may leverage Redis. Spring Cloud implementation uses **token bucket algorithm** to do rate limiting. This algorithm has a centralized bucket host where you take tokens on each request, and slowly drip more tokens into the bucket. If the bucket is empty, it rejects the request.

1. Dependencies

We will test our sample application against Spring Cloud Gateway rate limiting under higher traffic. First, we need to include some dependencies. Of course, Spring Cloud Gateway starter is required. For handling rate limiter with Redis we also need to add dependency to `spring-boot-starter-data-redis-reactive` starter. Other dependencies are used for test purposes. Module `mockserver` provided within Testcontainers. It is responsible for mocking a target service. In turn, the library `mockserver-client-java` is used for integration with `mockserver` container during the test. And the last library `junit-benchmarks` is used for a benchmarking test method and running the test concurrently.

```
<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.projectlombok</groupId>
   <artifactId>lombok</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>
   <version>1.12.3</version>
   <scope>test</scope>
```

```
</dependency>
<dependency>
   <groupId>org.mock-server</groupId>
   <artifactId>mockserver-client-java</artifactId>
   <version>3.10.8</version>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>com.carrotsearch</groupId>
   <artifactId>junit-benchmarks</artifactId>
   <version>0.7.2</version>
   <scope>test</scope>
</dependency>
```

The sample application is built on top of Spring Boot `2.2.1.RELEASE` and uses Spring Cloud `Hoxton.RC2` Release Train.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.2.1.RELEASE</version>
</parent>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
   <java.version>11</java.version>
   <spring-cloud.version>Hoxton.RC2</spring-cloud.version>
</properties>

<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>
```

2. Implementation

Request rate limiting is realized using a Spring Cloud Gateway component called `GatewayFilter`. Each instance of this filter is constructed in a specific factory. Filter is of course responsible for modifying requests and responses before or after sending the downstream request. Currently, there are 30 available built-in gateway filter factories.
The `GatewayFilter` takes an optional `keyResolver` parameter and parameters specific to the rate limiter implementation (in that case an implementation using Redis). Parameter `keyResolver` is a bean that implements the `KeyResolver` interface. It allows you to apply different strategies to derive the key for limiting requests. Parameter `keyResolver` is a bean that implements the `KeyResolver` interface. It allows you to apply different strategies to derive the key for limiting requests. Following Spring Cloud Gateway documentation:

> The default implementation of `KeyResolver` is the `PrincipalNameKeyResolver` which retrieves the `Principal` from the `ServerWebExchange` and calls `Principal.getName()`. By default, if the KeyResolver does not find a key, requests will be denied. This behavior can be adjusted with the `spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key` (true or false) and `spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code` properties.

Since, we have discussed some theoretical aspects of Spring Cloud Gateway rate limiting we may proceed to the implementation. First, let's define the main class and very simple `KeyResolver` bean, that is always equal to one.

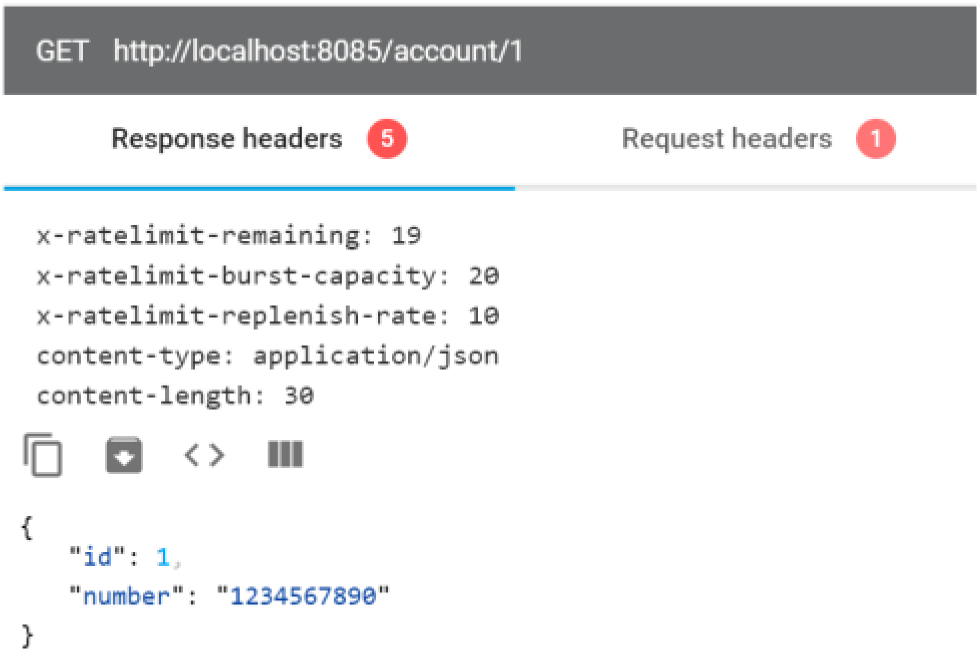
```
@SpringBootApplication
public class GatewayApplication {

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

   @Bean
   KeyResolver userKeyResolver() {
      return exchange -> Mono.just("1");
   }
}
```

Assuming we have the following configuration and a target application running on port `8091` we may perform some test calls. You may set two properties for customizing the process. The `redis-rate-limiter.replenishRate` decides how many requests per second a user is allowed to send without any dropped requests. This is the rate that the token bucket is filled. The second property `redis-rate-limiter.burstCapacity` is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero will block all requests.

```
server:
  port: ${PORT:8085}
```

```
spring:
  application:
    name: gateway-service
  redis:
    host: 192.168.99.100
    port: 6379
  cloud:
    gateway:
      routes:
      - id: account-service
        uri: http://localhost:8091
        predicates:
        - Path=/account/**
        filters:
        - RewritePath=/account/(?<path>.*), /$\{path}
      - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
```

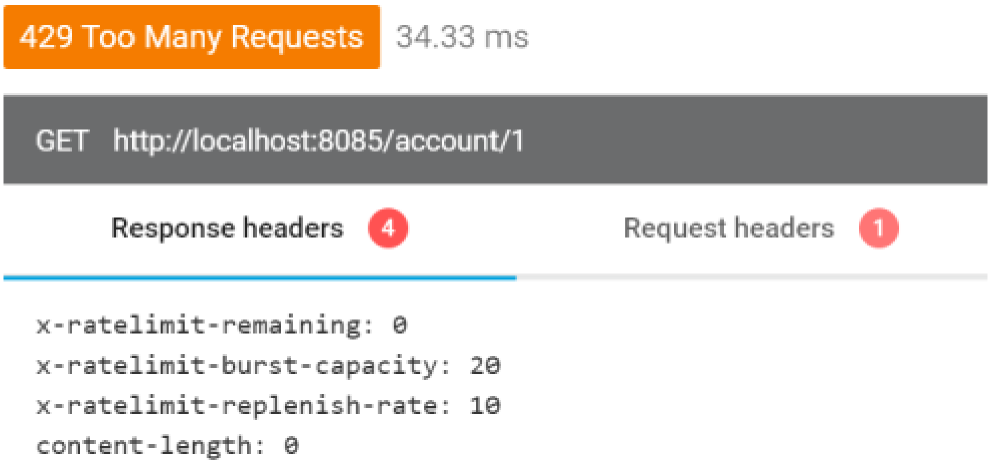
Now, if you call the endpoint exposed by the gateway you get the following response. It includes some specific headers, which are prefixed by `x-ratelimit`. Header `x-ratelimit-burst-capacity` indicates to `burstCapacity` value, `x-ratelimit-replenish-rate` indicates to `replenishRate` value, and the most important `x-ratelimit-remaining`, which shows you the number of requests you may send in the next second.



```
x-ratelimit-remaining: 19
x-ratelimit-burst-capacity: 20
x-ratelimit-replenish-rate: 10
content-type: application/json
content-length: 30
```

```
{
   "id": 1,
   "number": "1234567890"
}
```

If you exceed the number of allowed requests Spring Cloud Gateway return response with code `HTTP 429 - Too Many Requests`, and will not process the incoming request.



```
GET http://localhost:8085/account/1
```

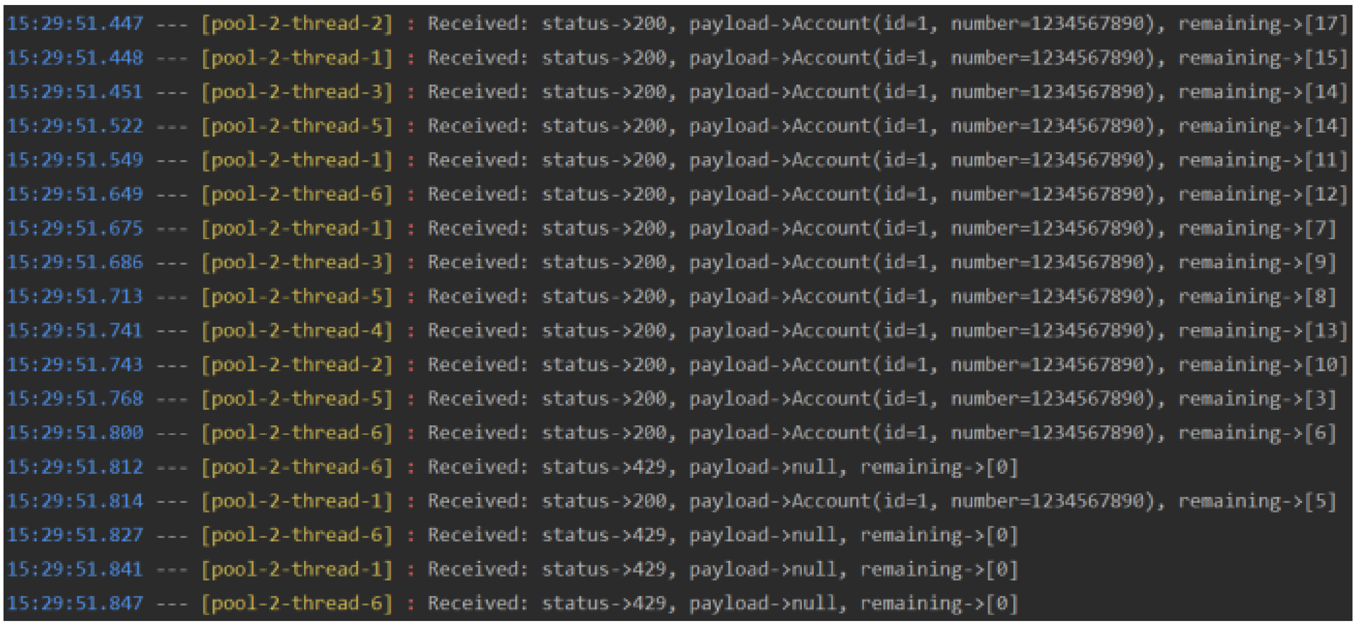
```
x-ratelimit-remaining: 0
x-ratelimit-burst-capacity: 20
x-ratelimit-replenish-rate: 10
content-length: 0
```

3. Testing Spring Cloud Gateway rate limiting

We have the Spring Boot test that uses two Docker containers provided by Testcontainers: `MockServer` and `Redis`. Because the exposed port is generated dynamically we need to set gateway properties in `@BeforeClass` method before running the test. Inside the init method we also use `MockServerClient` to define mock service on the mock server container. Our test method is running concurrently in six threads and is repeated 600 times.

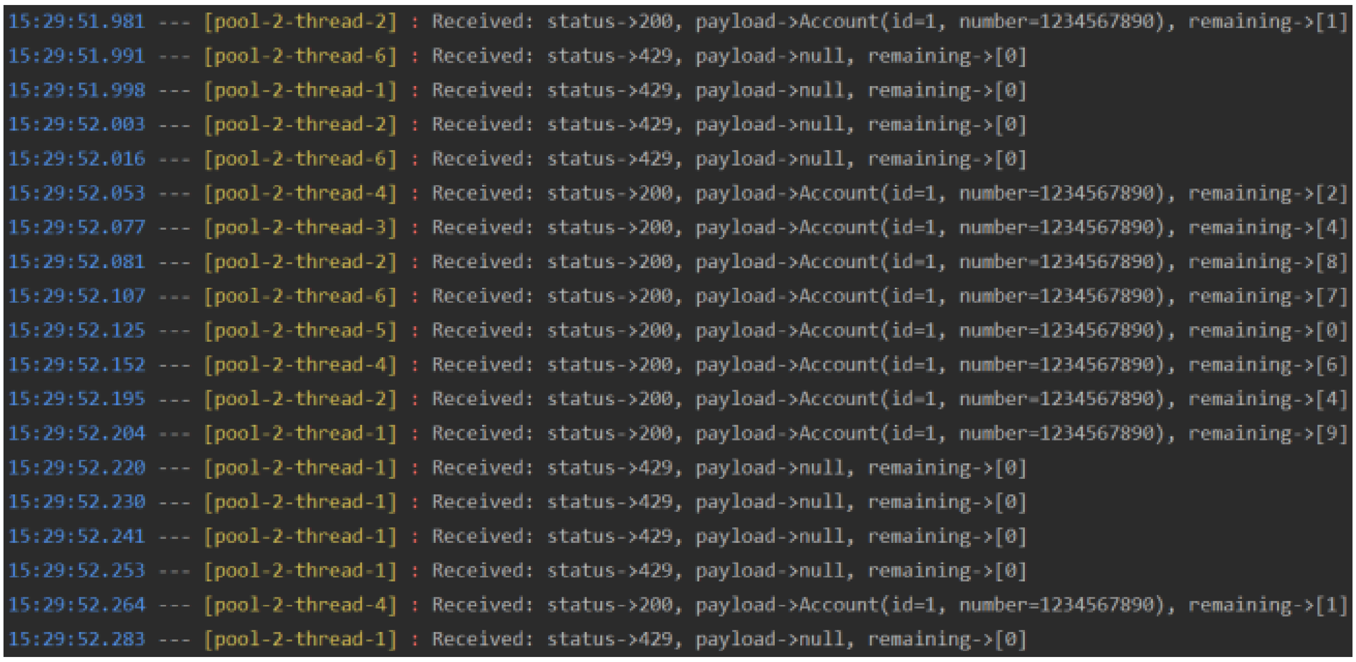
```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
@RunWith(SpringRunner.class)
public class GatewayRateLimiterTest {

    private static final Logger LOGGER = LoggerFactory.getLogger(GatewayRateLimiterTest.cla

    @Rule
    public TestRule benchmarkRun = new BenchmarkRule();

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

    @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://192.168.99.100:" +
        System.setProperty("spring.cloud.gateway.routes[0].predicates[0]", "Path=/account/*
        System.setProperty("spring.cloud.gateway.routes[0].filters[0]", "RewritePath=/accou
        System.setProperty("spring.cloud.gateway.routes[0].filters[1].name", "RequestRateLi
        System.setProperty("spring.cloud.gateway.routes[0].filters[1].args.redis-rate-limit
        System.setProperty("spring.cloud.gateway.routes[0].filters[1].args.redis-rate-limit
        System.setProperty("spring.redis.host", "192.168.99.100");
        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 = 6, benchmarkRounds = 600)
    public void testAccountService() {
        ResponseEntity<Account> r = template.exchange("/account/{id}", HttpMethod.GET, null
        LOGGER.info("Received: status->{}, payload->{}, remaining->{}", r.getStatusCodeValu
    }

}
```

Let's take a look at the test result. After starting gateway allows a user to send max 20 requests in a single second. After exceeding this value it starts to return `HTTP 429`.



```
15:29:51.447 --- [pool-2-thread-2] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[17]
15:29:51.448 --- [pool-2-thread-1] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[15]
15:29:51.451 --- [pool-2-thread-3] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[14]
15:29:51.522 --- [pool-2-thread-5] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[14]
15:29:51.549 --- [pool-2-thread-1] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[11]
15:29:51.649 --- [pool-2-thread-6] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[12]
15:29:51.675 --- [pool-2-thread-1] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[7]
15:29:51.686 --- [pool-2-thread-3] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[9]
15:29:51.713 --- [pool-2-thread-5] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[8]
15:29:51.741 --- [pool-2-thread-4] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[13]
15:29:51.743 --- [pool-2-thread-2] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[10]
15:29:51.768 --- [pool-2-thread-5] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[3]
15:29:51.800 --- [pool-2-thread-6] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[6]
15:29:51.812 --- [pool-2-thread-6] : Received: status->429, payload->null, remaining->[0]
15:29:51.814 --- [pool-2-thread-1] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[5]
15:29:51.827 --- [pool-2-thread-6] : Received: status->429, payload->null, remaining->[0]
15:29:51.841 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:51.847 --- [pool-2-thread-6] : Received: status->429, payload->null, remaining->[0]
```

After dropping some incoming requests gateway starts to accept them in the next second. But this time it allows to process only 10 requests, which is equal to `replenishRate` parameter value.


```
15:29:51.981 --- [pool-2-thread-2] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[1]
15:29:51.991 --- [pool-2-thread-6] : Received: status->429, payload->null, remaining->[0]
15:29:51.998 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:52.003 --- [pool-2-thread-2] : Received: status->429, payload->null, remaining->[0]
15:29:52.016 --- [pool-2-thread-6] : Received: status->429, payload->null, remaining->[0]
15:29:52.053 --- [pool-2-thread-4] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[2]
15:29:52.077 --- [pool-2-thread-3] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[4]
15:29:52.081 --- [pool-2-thread-2] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[8]
15:29:52.107 --- [pool-2-thread-6] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[7]
15:29:52.125 --- [pool-2-thread-5] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[0]
15:29:52.152 --- [pool-2-thread-4] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[6]
15:29:52.195 --- [pool-2-thread-2] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[4]
15:29:52.204 --- [pool-2-thread-1] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[9]
15:29:52.220 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:52.230 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:52.241 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:52.253 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
15:29:52.264 --- [pool-2-thread-4] : Received: status->200, payload->Account(id=1, number=1234567890), remaining->[1]
15:29:52.283 --- [pool-2-thread-1] : Received: status->429, payload->null, remaining->[0]
```

The **source code** is available in GitHub repository: **https://github.com/piomin/sample-spring-cloud-gateway.git**.

Like this:

rate limiting | Redis | Spring Cloud | Spring Cloud Gateway

Written by:

piotr.minkowski

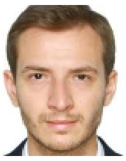
View All Posts →

21 COMMENTS

Chowdhury S Masood March 17, 2020 4:44 am

I am testing the rate limit feature using postman. I check the response header and see that X-RateLimit-Remaining is -1. Why is it negative? Of course, rate limit is not working.

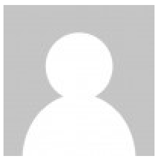
REPLY



Piotr Mińkowski March 30, 2020 5:30 pm

You should run JUnit test that performs such test. I don't understand what you mean by testing it with postman, since I implemented this mechanism only inside unit test, not at the level of application.

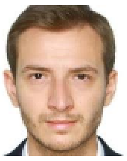
REPLY



Chowdhury S Masood March 31, 2020 6:08 pm

So, we are saying I could not use your ideas in a real application? I want to use the rate limit feature in a real application. What else I need to do? I trust JUnit test will pass.

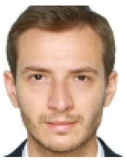
REPLY



Piotr Mińkowski April 28, 2020 10:29 pm

Hi. Take a look on my video course https://www.youtube.com/watch?v=XIkSWHX38Tg. You also have a repository with sample source code: https://github.com/piomin/course-spring-microservices.git.

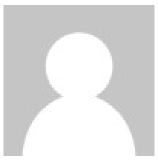
REPLY



Piotr Mińkowski March 31, 2020 6:13 pm

I didn't. I just don't know how did you run it. What parameters have you exactly set? how many requests did you send? Share your code with me maybe

REPLY

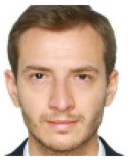


Bintoumadi April 13, 2020 2:22 pm

I tried to implement your solution. but each time I have: X-RateLimit-Remaining: -1.
and the following error in my console:
org.springframework.data.redis.RedisConnectionFailureException: Unable to connect to Redis; nested exception is io.lettuce.core.RedisConnectionException: Unable to connect to 127.0.0.1:6379

REPLY

Piotr Mińkowski April 28, 2020 9:58 pm



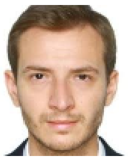
Did you run Redis?

REPLY

mohamed April 29, 2020 9:13 am

No. I just added the radish dependency to my project.

REPLY



Piotr Mińkowski April 29, 2020 7:31 am

In my test I used testcontainers to start Redis.

REPLY

Mohamed April 29, 2020 12:24 pm

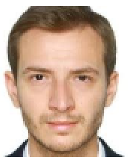
Ok. I think it's because I didn't run redis.
thank you for your help.

Devender Chaudhary May 3, 2020 6:52 pm

I tried integrating it with my project, but i am getting this error
java.lang.IllegalArgumentException: Unable to find GatewayFilterFactory with name RequestRateLimiter

Kind help if I am missing any thing.

REPLY



Piotr Mińkowski May 3, 2020 5:04 pm

Which version of Spring Cloud do you use in your project?

REPLY

Ahmed July 22, 2020 10:29 am

First of all thanks for this great tutorial, Piotr!
I have a question, can we increase replenishRate to be per 10 seconds or per minute or more based on the route? Or generally increase this rate?

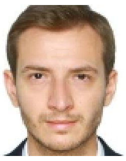
I tried to search a little bit and found that there is a redis lua script inside spring cloud gateway source code but I don't know lua and I am not sure that would be the way to go to edit the window of replenishRate.

REPLY

Sagar Gaikwad January 19, 2021 12:15 pm

How to set rate limit programmatically and not through properties?

REPLY



piotr.minkowski January 20, 2021 10:11 am

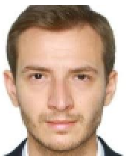
You can always create a `RouteLocator` bean and do it as visible below:
`@Bean public RouteLocator routes(RouteLocatorBuilder builder) { return builder.routes()... }`

REPLY

Jigar Prajapati August 24, 2021 9:40 am

Does it will save rate limit as key in redis ?

REPLY



piotr.minkowski August 31, 2021 3:24 pm

Yes

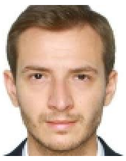
REPLY

Aylin January 3, 2022 11:09 am

i am getting exception in tests when i set key-resolver
key-resolver: "#{userKeyResolver}"

it throws this exception:
reactor.core.Exceptions$ErrorCallbackNotImplemented:
org.springframework.expression.spel.SpelEvaluationException: EL1007E: Property or field 'userKeyResolver' cannot be found on null
Caused by: org.springframework.expression.spel.SpelEvaluationException: EL1007E: Property or field 'userKeyResolver' cannot be found on null

REPLY



piotr.minkowski January 30, 2023 8:46 am

In that case you need to define key-resolver bean.

REPLY

Prateek January 13, 2023 9:52 am

Hi,
Could you share a link to other resources, it expects Account-service to be up, I dont see its code

```
io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused:
localhost/0:0:0:0:0:0:0:1:8091
Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Error has been observed at the following site(s):
*__checkpoint ⇢ org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter
[DefaultWebFilterChain]
```

★ Loading...

REPLY

piotr.minkowski January 18, 2023 11:27 pm

Because there is no account-service. It is mocked. Here's the full code: https://github.com/piomin/sample-spring-cloud-gateway/blob/master/src/test/java/pl/piomin/services/gateway/GatewayRateLimiterTest.java

★ Loading...

REPLY