**RateLimiter**

**RateLimiter of API using Reselience4j**

Resilience4j is a lightweight, modular, and fault-tolerant library designed for Java 8 and functional programming. It provides a set of modules for building resilient applications, helping developers handle failures and transient errors in a more systematic way.

The RateLimiter module helps control the rate at which requests are made to a particular resource. It can be useful for limiting the load on downstream services.

Demo Project was done to illustrate the RateLimiter using Reselience4j.

**Dependencies Used:**

<dependency>

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

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

<version>2.2.6. RELEASE</version>

</dependency>

<dependency>

<groupId>io.github. resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

<version>1.4.0</version>

</dependency>

<dependency>

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

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

<version>2.2.1. RELEASE</version>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

**OrderController class**

RestTemplate is a widely used class in the Spring Framework for making HTTP requests to external services or APIs.In our application we used RestTemplate inorder to call the API of another controller ie ItemController.We called the API of ItemController using getForObject(ResponseEntity as object) method. Before calling we specified the response should be of String Class. The number of API hits will be restricted to this method after the specified time. If the requests are given after the specified time or if any exception is thrown, then fallback method is called.

*@Bean*

*public RestTemplate getRestTemplate() {*

*return new RestTemplate();*

*}*

*@Autowired*

*private RestTemplate restTemplate;*

*@GetMapping("/order")*

*@RateLimiter (name=ORDER\_SERVICE, fallbackMethod = "rateLimiterFallback")*

*public ResponseEntity<String> createOrder()*

*{*

*String response = restTemplate.getForObject("http://localhost:8080/item", String.class);*

*logger.info (LocalTime.now() + " Call processing finished = " + Thread.currentThread().getName());*

*return new ResponseEntity<String> (response, HttpStatus.OK);*

*}*

*public ResponseEntity<String> rateLimiterFallback(Exception e){*

*return new ResponseEntity<String> ("order service does not permit further calls", HttpStatus.TOO\_MANY\_REQUESTS);*

*}*

*}*

**ItemController Class**

*private static final Logger logger = LoggerFactory.getLogger(ItemController.class);*

*@GetMapping("/item")*

*public String getItem() {*

*logger.info ("GetItem call returned");*

*return "Item Selected successfully";*

*}*

**ApplicationProperties file**

Only two threads will execute at a time. After the 4 seconds all the requests will be rejected.The response will be “order service does not permit further calls”

*resilience4j:*

*ratelimiter:*

*instances:*

*orderService:*

*limitForPeriod: 2*

*limitRefreshPeriod: 4s*

*timeoutDuration: 0s*

*registerHealthIndicator: true*

*eventConsumerBufferSize: 100*

*management:*

*endpoints:*

*web:*

*exposure:*

*include: '\*'*

*endpoint:*

*health:*

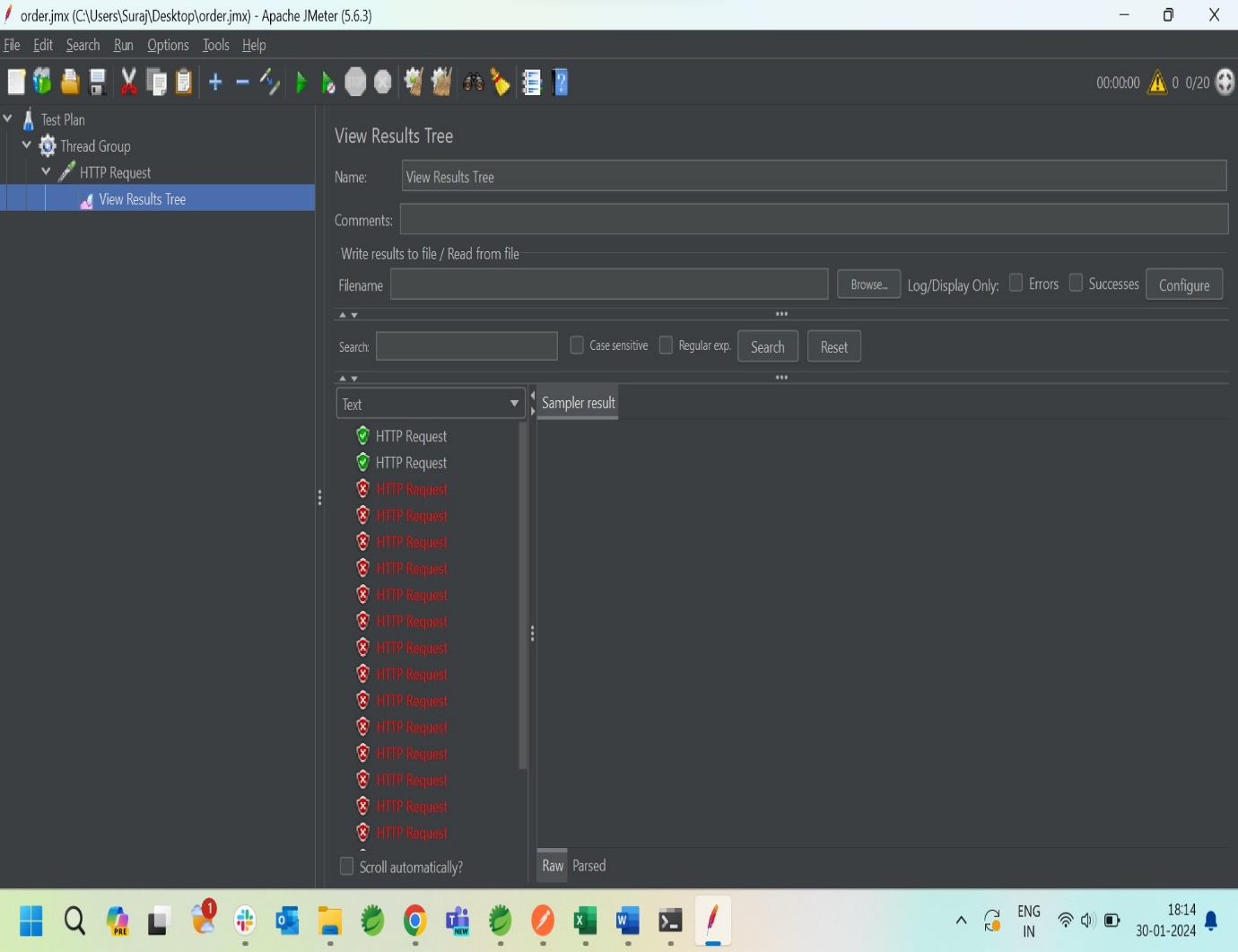
*show-details: always*

We monitored with threads running using JMeter tool. Only two threads were executing at a time. After the 4s, all the requests were rejected.

**JMeter Tool**

**Steps for Montitoring**

* Go to Threads
* Add ThreadGroup
* Give the total number of Threads you want to execute.
* Go to Listener
* View Results Tree
* Run the Springboot application in Background.
* Click on Run in JMeter
* We will get the number of threads executing for the given time.



**RateLimiter based on IP**

Demo project was done to illustrate the RateLimiter based on IP

**Dependencies used for the Project:**

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>31.0-jre</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

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

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

<scope>test</scope></dependency>

**RateLimitConfig class**

Here we are creating the list of urls that should be allowed during specific time frame and also we are allowing the maximum of 3 requests. The requestCountsPerIpAddress stores the number of api requests per IP address or else it will take the client-hash-id as a parameter. For getting the client ip we have written the getClientIP method.

*public class RateLimitConfig implements Filter {*

*private int MAX\_REQUEST = 3;*

*/\*\**

*\* The key-value pairs in the cache represent IP addresses and their corresponding request counts.*

*\*/*

*private LoadingCache<String,Integer> requestCountsPerIpAddress;*

*private Set<String> urls = new HashSet<> (Arrays.asList(*

*"/api/v1/rate-limit",*

*"/api/v1/rate-limit/two"*

*));*

Here, we are taking the the input urls from the request and store it in finalUri. If the urls contains the final uri, then it will go in the if condition. We have created a client-hash-id which we will pass through the request header during api request. We can pass the client-hash-d or the IP address in the header. If the request-header is empty, then it will take the ip of the client. For getting the ip of the client, we have written the getClientIP() method.

*public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {*

*HttpServletResponse httpServletResponse = (HttpServletResponse) servletResponse;*

*HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;*

*String some = httpServletRequest.getRequestURI();*

*System.out.println("url = " + some);*

*String finalUri = (httpServletRequest.getRequestURI(). substring(httpServletRequest.getContextPath().length()));*

*if(urls.contains(finalUri)) {*

*/\*\* clientHashId contains url+/ [clientHashId or IP] \*/*

*String clientHashId;*

*if (StringUtils.isEmpty(httpServletRequest.getHeader("client-hash-id")))*

*clientHashId = finalUri+"/"+getClientIP((HttpServletRequest) servletRequest);*

*else clientHashId = finalUri+"/"+httpServletRequest.getHeader("client-hash-id");*

*System.out.println("clientHashId = " + clientHashId);*

*// System.out.println(clientHashId);*

*if (isMaxRequestsPerSecond(clientHashId)) {*

*httpServletResponse.setStatus(HttpStatus.TOO\_MANY\_REQUESTS.value());*

*httpServletResponse.getWriter().print(ResponseEntity.status(HttpStatus.TOO\_MANY\_REQUESTS).build());*

*return;*

*}*

*}*

*filterChain.doFilter(servletRequest,servletResponse);*

*}*

We will get the clientIP with is method using the key “X-Forwarded-For”. If the ip is null, The remote ip will be fetched.

*private String getClientIP(HttpServletRequest servletRequest) {*

*String ip = servletRequest.getHeader("X-Forwarded-For");*

*if (ip == null)*

*{*

*/\*\* Get the ipAddress of the client \*/*

*ip = servletRequest.getRemoteAddr();*

*return ip;*

*}*

*return ip.split(",")[0];*

*}*

Here we are checking the number of API requests has been exceeded or not in this method. The API requests per client-hash-id is stored in requestCountsPerIpAddress and we are incrementing the req as soon as the API are hit. If the req is greater than MAX\_REQUEST, then “EXCEEDED” will be executed.

*private boolean isMaxRequestsPerSecond(String clientHashId) {*

*int req =0;*

*try {*

*/\*\**

*Fetching the number of requests using the key*

*\*/*

*req = requestCountsPerIpAddress.get(clientHashId);*

*System.out.println("req = " + req);*

*// System.out.println(req);*

*if(req>MAX\_REQUEST)*

*{*

*System.out.println("EXCEEDED = " + new Date ());*

*requestCountsPerIpAddress.put(clientHashId,req);*

*return true;*

*}*

*}*

*catch (Exception e) {*

*req =0;*

*}*

*req++;*

*requestCountsPerIpAddress.put(clientHashId,req);*

*return false;*

*}*

**RateLimiter using Bucket4j.**

Bucket4j is a Java rate-limiting library based on the token-bucket algorithm. Bucket4j is a thread-safe library that can be used in either a standalone JVM application, or a clustered environment. It also supports in-memory or distributed caching via the JCache (JSR107) specification.

**Demo Project was done to illustrate RateLimiter using Bucket4j.**

**Dependencies used:**

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-web-services</artifactId>

</dependency>

<dependency>

<groupId>com.github.vladimir-bukhtoyarov</groupId>

<artifactId>bucket4j-core</artifactId>

<version>7.6.0</version>

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

RateLimiterController Class

In this context:

The tokens represent the permission or allowance for API hits.

Each successful API hit consumes one token from the bucket.

If the bucket is empty (no tokens available), it means that the rate limit has been exceeded.

The rate limit is defined by the Refill object, which specifies how many tokens are refilled and the duration of the refill. In your code, it's set to refill 5 tokens every 1 minute.

Token Generation (/tokengenerate endpoint):

When this endpoint is accessed (/ratelimit/tokengenerate), a new Bucket is created with a rate limit of 5 tokens per minute.

This essentially means that the bucket is being configured to allow a maximum of 5 API hits within a 1-minute window.

*@GetMapping("/tokengenerate")*

*public ResponseEntity<String> generateToken() {*

*// refill*

*Refill refill = Refill.of(5, Duration.ofMinutes(1)); //create 5 tokens per minute.For each api hit,one token will be consumed.*

*// bucket*

*bucket = Bucket4j.builder().addLimit(Bandwidth.classic(5, refill)) // Limited to 5*

*.build();*

*System.out.println(bucket);*

*return new ResponseEntity<String>("Generated Successfully!!", HttpStatus.OK);*

*}*

API Hit Check (/demo endpoint):

The /demo endpoint checks if the bucket can "consume" a token when an API hit occurs.

If the bucket allows consumption (there is at least one token available), the API hit is considered successful, and the response is "SUCCESS."

If the bucket does not allow consumption (no tokens available), the API hit is considered as exceeding the rate limit, and the response is "TOO MANY HITS!!!!" with a 429 (TOO\_MANY\_REQUESTS) HTTP status.

*@GetMapping("/demo")*

*public ResponseEntity<String> demo() {*

*if (bucket.tryConsume(1))// If it consumes,then there is atleast one token available.*

*{*

*System.out.println("==============API working successfully=========");*

*return new ResponseEntity<String>("SUCCESS", HttpStatus.OK);*

*}*

*System.out.println("==============Number of Hits Exceeded=========");*

*return new ResponseEntity<String>("TOO MANY HITS!!!!", HttpStatus.TOO\_MANY\_REQUESTS);*

*}*