Rate Limiting a Spring API Using Bucket4j

Last modified: December 10, 2022

Written by: Priyank Srivastava

REST Spring Boot

1. Overview

In this tutorial, **we'll focus on how to use Bucket4j to rate limit a Spring REST API**.

We'll explore API rate limiting, learn about Bucket4j, and then work through a few ways of rate-limiting REST APIs in a Spring appli

2. API Rate Limiting

Rate limiting is a strategy to limit access to APIs. It restricts the number of API calls that a client can make within a certain time frar

Rate limits are often applied to an API by tracking the IP address, or in a more business-specific way, such as API keys or access tc

- Queueing the request until the remaining time period has elapsed
- Allowing the request immediately, but charging extra for this request
- Rejecting the request (HTTP 429 Too Many Requests)

3. Bucket4j Rate Limiting Library

3.1. What Is 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 ei

3.2. Token-bucket Algorithm

Let's look at the algorithm intuitively in the context of API rate limiting.

Say we have a bucket whose capacity is defined as the number of tokens that it can hold. **Whenever a consumer wants to acces**

As requests are consuming tokens, we're also replenishing them at some fixed rate, such that we never exceed the capacity of

Let's consider an API that has a rate limit of 100 requests per minute. We can create a bucket with a capacity of 100, and a refill rat

If we receive 70 requests, which is fewer than the available tokens in a given minute, we would add only 30 more tokens at the sta

4. Getting Started With Bucket4j

4.1. Maven Configuration

Let's begin by adding the *bucket4j* dependency to our *pom.xml*:

```
<dependency>
    <groupId>com.github.vladimir-bukhtoyarov</groupId>
    <artifactId>bucket4j-core</artifactId>
    <version>7.6.0</version>
</dependency>
```

4.2. Terminology

Before we look at how to use Bucket4j, we'll briefly discuss some of the core classes, and how they represent the different elemer

The *Bucket* interface represents the token bucket with a maximum capacity. It provides methods such as *tryConsume* and *tryCon*

The *Bandwidth* class is the key building block of a bucket, as it defines the limits of the bucket. We use *Bandwidth* to configure the

The *Refill* class is used to define the fixed rate at which tokens are added to the bucket. We can configure the rate as the number

The *tryConsumeAndReturnRemaining* method in *Bucket* returns *ConsumptionProbe*. *ConsumptionProbe* contains, along with the

4.3. Basic Usage

Let's test some basic rate limit patterns.

For a rate limit of 10 requests per minute, we'll create a bucket with capacity 10 and a refill rate of 10 tokens per minute:

```
Refill refill = Refill.intervally(10, Duration.ofMinutes(1));
Bandwidth limit = Bandwidth.classic(10, refill);
Bucket bucket = Bucket.builder()
    .addLimit(limit)
    .build();

for (int i = 1; i <= 10; i++) {
    assertTrue(bucket.tryConsume(1));
}
assertFalse(bucket.tryConsume(1));
```

Refill.intervally refills the bucket at the beginning of the time window, which in this case is 10 tokens at the start of the minute.

Next, let's see refill in action.

We'll set a refill rate of 1 token per 2 seconds, and **throttle our requests to honor the rate limit**:

```
Bandwidth limit = Bandwidth.classic(1, Refill.intervally(1, Duration.ofSeconds(2)));
Bucket bucket = Bucket.builder()
    .addLimit(limit)
    .build();
assertTrue(bucket.tryConsume(1));     // first request
Executors.newScheduledThreadPool(1)   // schedule another request for 2 seconds later
    .schedule(() -> assertTrue(bucket.tryConsume(1)), 2, TimeUnit.SECONDS);
```

Suppose we have a rate limit of 10 requests per minute. At the same time, **we may wish to avoid spikes that would exhaust all th**

```
Bucket bucket = Bucket.builder()
    .addLimit(Bandwidth.classic(10, Refill.intervally(10, Duration.ofMinutes(1))))
    .addLimit(Bandwidth.classic(5, Refill.intervally(5, Duration.ofSeconds(20))))
    .build();

for (int i = 1; i <= 5; i++) {
    assertTrue(bucket.tryConsume(1));
}
assertFalse(bucket.tryConsume(1));
```

5. Rate Limiting a Spring API Using Bucket4j

Let's use Bucket4j to apply a rate limit in a Spring REST API.

5.1. Area Calculator API

We'll implement a simple, but extremely popular, area calculator REST API. Currently, it calculates and returns the area of a rectan

```
@RestController
class AreaCalculationController {

    @PostMapping(value = "/api/v1/area/rectangle")
    public ResponseEntity<AreaV1> rectangle(@RequestBody RectangleDimensionsV1 dimensions) {
        return ResponseEntity.ok(new AreaV1("rectangle", dimensions.getLength() * dimensions.getWidth()));
    }
}
```

Let's ensure that our API is up and running:

```
$ curl -X POST http://localhost:9001/api/v1/area/rectangle \
    -H "Content-Type: application/json" \
    -d '{ "length": 10, "width": 12 }'

{ "shape":"rectangle","area":120.0 }
```

5.2. Applying Rate Limit

Now we'll introduce a naive rate limit, allowing the API 20 requests per minute. In other words, the API rejects a request if it's alrea

Let's modify our *Controller* to create a *Bucket* and add the limit *(Bandwidth):*

```
@RestController
class AreaCalculationController {

    private final Bucket bucket;

    public AreaCalculationController() {
        Bandwidth limit = Bandwidth.classic(20, Refill.greedy(20, Duration.ofMinutes(1)));
        this.bucket = Bucket.builder()
            .addLimit(limit)
            .build();
    }
    //..
}
```

In this API, we can check whether the request is allowed by consuming a token from the bucket using the method *tryConsume*. If

```
public ResponseEntity<AreaV1> rectangle(@RequestBody RectangleDimensionsV1 dimensions) {
    if (bucket.tryConsume(1)) {
        return ResponseEntity.ok(new AreaV1("rectangle", dimensions.getLength() * dimensions.getWidth()));
    }

    return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS).build();
}
```

```
# 21st request within 1 minute
$ curl -v -X POST http://localhost:9001/api/v1/area/rectangle \
    -H "Content-Type: application/json" \
    -d '{ "length": 10, "width": 12 }'

< HTTP/1.1 429
```

5.3. API Clients and Pricing Plan

Now we have a naive rate limit that can throttle the API requests. Next, we'll introduce pricing plans for more business-centered ra

Pricing plans help us monetize our API. Let's assume that we have the following plans for our API clients:

- Free: 20 requests per hour per API client
- Basic: 40 requests per hour per API client
- Professional: 100 requests per hour per API client

Each API client gets a **unique API key that they must send along with each request**. This helps us identify the pricing plan linked

Let's define the rate limit (*Bandwidth*) for each pricing plan:

```
enum PricingPlan {
    FREE {
        Bandwidth getLimit() {
            return Bandwidth.classic(20, Refill.intervally(20, Duration.ofHours(1)));
        }
    },
    BASIC {
        Bandwidth getLimit() {
            return Bandwidth.classic(40, Refill.intervally(40, Duration.ofHours(1)));
        }
    },
    PROFESSIONAL {
        Bandwidth getLimit() {
            return Bandwidth.classic(100, Refill.intervally(100, Duration.ofHours(1)));
        }
    };
    //..
}
```

Then let's add a method to resolve the pricing plan from the given API key:

```
enum PricingPlan {

    static PricingPlan resolvePlanFromApiKey(String apiKey) {
        if (apiKey == null || apiKey.isEmpty()) {
            return FREE;
        } else if (apiKey.startsWith("PX001-")) {
            return PROFESSIONAL;
        } else if (apiKey.startsWith("BX001-")) {
            return BASIC;
        }
        return FREE;
    }
    //..
}
```

Next, we need to store the *Bucket* for each API key, and retrieve the *Bucket* for rate limiting:

```
class PricingPlanService {

    private final Map<String, Bucket> cache = new ConcurrentHashMap<>();

    public Bucket resolveBucket(String apiKey) {
        return cache.computeIfAbsent(apiKey, this::newBucket);
    }

    private Bucket newBucket(String apiKey) {
        PricingPlan pricingPlan = PricingPlan.resolvePlanFromApiKey(apiKey);
        return Bucket.builder()
            .addLimit(pricingPlan.getLimit())
            .build();
    }
}
```

Now we have an in-memory store of buckets per API key. Let's modify our *Controller* to use the *PricingPlanService*:

```
@RestController
class AreaCalculationController {

    private PricingPlanService pricingPlanService;

    public ResponseEntity<AreaV1> rectangle(@RequestHeader(value = "X-api-key") String apiKey,
        @RequestBody RectangleDimensionsV1 dimensions) {

        Bucket bucket = pricingPlanService.resolveBucket(apiKey);
        ConsumptionProbe probe = bucket.tryConsumeAndReturnRemaining(1);
        if (probe.isConsumed()) {
            return ResponseEntity.ok()
                .header("X-Rate-Limit-Remaining", Long.toString(probe.getRemainingTokens()))
                .body(new AreaV1("rectangle", dimensions.getLength() * dimensions.getWidth()));
        }

        long waitForRefill = probe.getNanosToWaitForRefill() / 1_000_000_000;
        return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS)
            .header("X-Rate-Limit-Retry-After-Seconds", String.valueOf(waitForRefill))
            .build();
    }
}
```

Let's walk through the changes. The API client sends the API key with the *X-api-key* request header. We use the *PricingPlanServic*

In order to enhance the client experience of the API, we'll use the following additional response headers to send information abou

- *X-Rate-Limit-Remaining*: number of tokens remaining in the current time window
- *X-Rate-Limit-Retry-After-Seconds*: remaining time, in seconds, until the bucket is refilled

We can call the *ConsumptionProbe* methods *getRemainingTokens* and *getNanosToWaitForRefill* to get the count of remaining tok

Let's call the API:

```
## successful request
$ curl -v -X POST http://localhost:9001/api/v1/area/rectangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "length": 10, "width": 12 }'

< HTTP/1.1 200
< X-Rate-Limit-Remaining: 11
{"shape":"rectangle","area":120.0}

## rejected request
$ curl -v -X POST http://localhost:9001/api/v1/area/rectangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "length": 10, "width": 12 }'

< HTTP/1.1 429
< X-Rate-Limit-Retry-After-Seconds: 583
```

5.4. Using Spring MVC Interceptor

Suppose we now have to add a new API endpoint that calculates and returns the area of a triangle given its height and base:

```
@PostMapping(value = "/triangle")
public ResponseEntity<AreaV1> triangle(@RequestBody TriangleDimensionsV1 dimensions) {
    return ResponseEntity.ok(new AreaV1("triangle", 0.5d * dimensions.getHeight() * dimensions.getBase()));
}
```

As it turns out, we need to rate-limit our new endpoint as well. We can simply copy and paste the rate limit code from our previou

Let's create a *RateLimitInterceptor* and implement the rate limit code in the *preHandle* method:

```java
public class RateLimitInterceptor implements HandlerInterceptor {

    private PricingPlanService pricingPlanService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
      throws Exception {
        String apiKey = request.getHeader("X-api-key");
        if (apiKey == null || apiKey.isEmpty()) {
            response.sendError(HttpStatus.BAD_REQUEST.value(), "Missing Header: X-api-key");
            return false;
        }

        Bucket tokenBucket = pricingPlanService.resolveBucket(apiKey);
        ConsumptionProbe probe = tokenBucket.tryConsumeAndReturnRemaining(1);
        if (probe.isConsumed()) {
            response.addHeader("X-Rate-Limit-Remaining", String.valueOf(probe.getRemainingTokens()));
            return true;
        } else {
            long waitForRefill = probe.getNanosToWaitForRefill() / 1_000_000_000;
            response.addHeader("X-Rate-Limit-Retry-After-Seconds", String.valueOf(waitForRefill));
            response.sendError(HttpStatus.TOO_MANY_REQUESTS.value(),
              "You have exhausted your API Request Quota");
            return false;
        }
    }
}
```

Finally, we must add the interceptor to the *InterceptorRegistry*:

```java
public class AppConfig implements WebMvcConfigurer {

    private RateLimitInterceptor interceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(interceptor)
            .addPathPatterns("/api/v1/area/**");
    }
}
```

The *RateLimitInterceptor* intercepts each request to our area calculation API endpoints.

Let's try out our new endpoint:

```
## successful request
$ curl -v -X POST http://localhost:9001/api/v1/area/triangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "height": 15, "base": 8 }'

< HTTP/1.1 200
< X-Rate-Limit-Remaining: 9
{"shape":"triangle","area":60.0}

## rejected request
$ curl -v -X POST http://localhost:9001/api/v1/area/triangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "height": 15, "base": 8 }'

< HTTP/1.1 429
< X-Rate-Limit-Retry-After-Seconds: 299
{ "status": 429, "error": "Too Many Requests", "message": "You have exhausted your API Request Quota" }
```

It looks like we're done. We can keep adding endpoints, and the interceptor will apply the rate limit for each request.

6. Bucket4j Spring Boot Starter

Let's look at another way of using Bucket4j in a Spring application. The Bucket4j Spring Boot Starter provides auto-configuration fo

Once we integrate the Bucket4j starter into our application, **we'll have a completely declarative API rate limiting implementatio**

6.1. Rate Limit Filters

In our example, we used the value of the request header *X-api-key* as the key for identifying and applying the rate limits.

The Bucket4j Spring Boot Starter provides several predefined configurations for defining our rate limit key:

- a naive rate limit filter, which is the default
- filter by IP Address
- expression-based filters

Expression-based filters use the Spring Expression Language (SpEL). SpEL provides access to root objects, such as *HttpServletRe*

The library also supports custom classes in the filter expressions, which is discussed in the documentation.

6.2. Maven Configuration

Let's begin by adding the *bucket4j-spring-boot-starter* dependency to our *pom.xml*:

```
<dependency>
    <groupId>com.giffing.bucket4j.spring.boot.starter</groupId>
    <artifactId>bucket4j-spring-boot-starter</artifactId>
    <version>0.7.0</version>
</dependency>
```

We used an in-memory *Map* to store the *Bucket* per API key (consumer) in our earlier implementation. Here, we can use Spring's c

Let's add the caching dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
</dependency>
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
    <version>2.8.2</version>
</dependency>
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>jcache</artifactId>
    <version>2.8.2</version>
</dependency>
```

Note: We added the *jcache* dependencies as well, to conform with Bucket4j's caching support.

We must remember to **enable the caching feature by adding the *@EnableCaching* annotation to any of the configuration class**

6.3. Application Configuration

Let's configure our application to use the Bucket4j starter library. First, we'll configure Caffeine caching to store the API key and *Bu*

```
spring:
  cache:
    cache-names:
    - rate-limit-buckets
    caffeine:
      spec: maximumSize=100000,expireAfterAccess=3600s
```

Next, let's configure Bucket4j:

```
bucket4j:
  enabled: true
  filters:
  - cache-name: rate-limit-buckets
    url: /api/v1/area.*
    strategy: first
    http-response-body: "{ \"status\": 429, \"error\": \"Too Many Requests\", \"message\": \"You have exhausted your API Reque
    rate-limits:
    - expression: "getHeader('X-api-key')"
      execute-condition: "getHeader('X-api-key').startsWith('PX001-')"
      bandwidths:
      - capacity: 100
        time: 1
        unit: hours
    - expression: "getHeader('X-api-key')"
      execute-condition: "getHeader('X-api-key').startsWith('BX001-')"
      bandwidths:
      - capacity: 40
        time: 1
        unit: hours
    - expression: "getHeader('X-api-key')"
      bandwidths:
      - capacity: 20
        time: 1
        unit: hours
```

So, what did we just configure?

- *bucket4j.enabled=true* – enables Bucket4j auto-configuration
- *bucket4j.filters.cache-name* – gets the *Bucket* for an API key from the cache
- *bucket4j.filters.url* – indicates the path expression for applying the rate limit
- *bucket4j.filters.strategy=first* – stops at the first matching rate limit configuration
- *bucket4j.filters.rate-limits.expression* – retrieves the key using Spring Expression Language (SpEL)
- *bucket4j.filters.rate-limits.execute-condition* – decides whether to execute the rate limit or not using SpEL
- *bucket4j.filters.rate-limits.bandwidths* – defines the Bucket4j rate limit parameters

We replaced the *PricingPlanService* and the *RateLimitInterceptor* with a list of rate limit configurations that are evaluated sequent

Let's try it out:

```
## successful request
$ curl -v -X POST http://localhost:9000/api/v1/area/triangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "height": 20, "base": 7 }'

< HTTP/1.1 200
< X-Rate-Limit-Remaining: 7
{"shape":"triangle","area":70.0}

## rejected request
$ curl -v -X POST http://localhost:9000/api/v1/area/triangle \
    -H "Content-Type: application/json" -H "X-api-key:FX001-99999" \
    -d '{ "height": 7, "base": 20 }'

< HTTP/1.1 429
< X-Rate-Limit-Retry-After-Seconds: 212
{ "status": 429, "error": "Too Many Requests", "message": "You have exhausted your API Request Quota" }
```

7. Conclusion

In this article, we demonstrated several different approaches using Bucket4j for rate-limiting Spring APIs. To learn more, be sure tc

As usual, the source code for all the examples is available over on GitHub.

4 COMMENTS

Oldest

Comments are closed on this article!