API RATE LIMIT

Problem Statement:

To build a server-side rate-limiting solution, which tracks the number of
requests that are made by a User over a given period of time and block requests when the number of calls exceeds the predefined threshold.

Analysis:

There are various algorithms for rate limiting:

1. Token Bucket Algorithm-: This could be considered as a few tokens in a bucket. When a request comes, a token has to be taken from the bucket for it to be processed. If there is no token available in the bucket, the request will be rejected.
   For each unique user, we would record their last request's Unix timestamp and available token count.Whenever a new request arrives from a user, It would fetch the hash.and refill the available tokens based on a chosen refill rate and the time of the user's last request. Then, it would update the hash with the current request's timestamp and the new available token count. When the available token count drops to zero, the rate limiter knows the user has exceeded the limit.Token Bucket allows traffic burst by nature. If tokens are not taken during some period of time, token bucket is refilled partly or completely. More tokens in a bucket, higher traffic burst allowed.

2. Fixed Window Algorithm-: Fixed window counter algorithm divides the timeline into fixed-size windows and assign a counter to each window. Each request, based on its arriving time, is mapped to a window. If the counter in the window has reached the limit, requests falling in this window should be rejected.When incrementing the request count for a given timestamp, we would compare its value to our rate limit to decide whether to reject the request. Fixed window counter algorithm only guarantees the average rate within each window but not across windows.

3. Sliding Window Algorithm-: In Sliding Window Counter Algorithm we increment counters specific to the current Unix minute timestamp and calculate the sum of all counters in the past hour when we receive a new request.When each request increments a counter in the hash, it also sets the hash to expire according to rate limit. In the event that a user makes requests every minute, the user's hash can grow large from holding onto counters for bygone timestamps. We prevent this by regularly removing these counters.

Note-: A Single Node application can implement rate limiter using in-memory cache while for multi node, we'll be needing some distributed cache that can be shared among different nodes. To take into the consideration of concurrent requests, we can use any thread-safe mechanism/thread-safe collection for in-memory cache, while redis can be used as distributed cache as it provides redis-lock to prevent access to concurrent requests.

Components:

1. Database: I have defined one table named with ‘rate_limiter_rules’, It will contains all the rules that have been provided by admin.
   Following is the structure for the same:

   id -: Autogenrated id/primary for a particular rule
   user_Id-: an identifier that is unique for every user
   api-: API Indentifiers(already defined in Application)
   rate_limit-: rate limit for that rule
   rate_limiter_time_unit-: Time unit for that rate limit (HOUR/MINUTE/SECOND)
   is_active-: flag to determine rule is active or not

ID	API	IS_ACTIVE	RATE_LIMIT	RATE_LIMITER_TIME_UNIT	USER_ID
1	API_GET_DEVELOPER_ALL	TRUE	10	MINUTE	user112
2	API_GET_ORGANIZATION_ALL	FALSE	10	SECOND	user114
3	API_GET_DEVELOPER_ONE	TRUE	10	SECOND	user116

2. Redis Server: In case of multiple nodes of application being deployed, we will be using distrubuted cache to implement ratelimiter algorithm.

3. Application: It contains the implementation for different algorithms for Rate Limiting Solution.

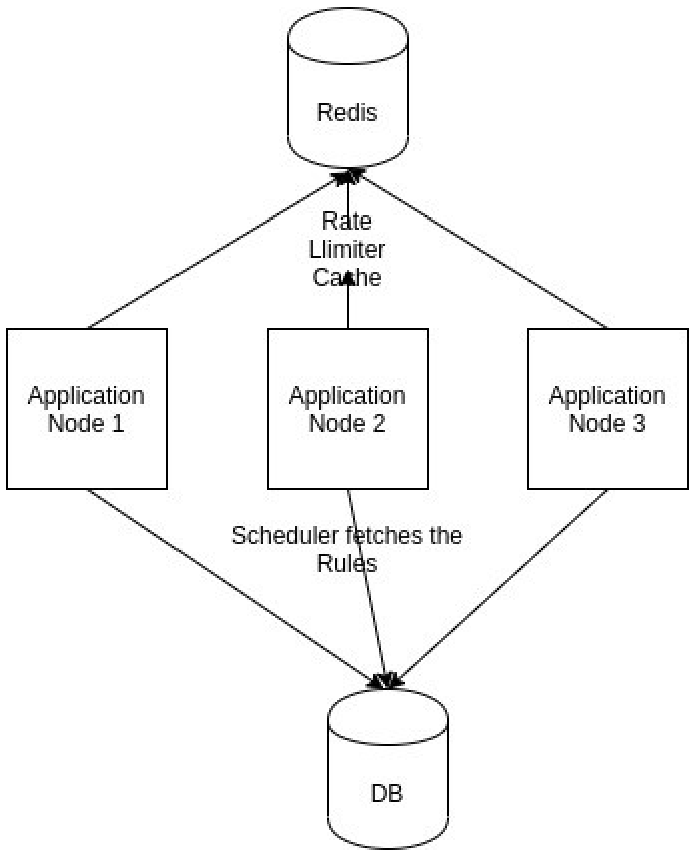
   Working:

   When application is started, a job is scheduled to run in background that fetches the data from rate_limiter_rule table, transformes that data into key:value pair and stores in in-memory cache in the application itself.

   It refreshes the cache after some interval that is provided in the configuration.

   When someone hits a particular API, a particular rule is fetched from cache according to combination of userID(provided in header) and API being called. According to the rule and the configuration provided, a specific rate limiter cache implementation along with the specific rate limiter algorithm is called.

Rate Limiter Basic Architecture:



Flows:

Note: Right now, H2 memory Db is used for testing purposes, later it can be changed to SQL DB that is deployed in production.

1. Add a new Rule:
   Curl Request:-

```
curl --location --request POST 'http://localhost:8080/rateLimiterRule' \
--header 'Content-Type: application/json' \
--data-raw ' {
        "userId": "user114",
        "api": "API_GET_DEVELOPER_ALL",
        "rateLimit": 10,
        "rateLimiterTimeUnit": "MINUTE",
        "active": true
 }'
```

2. GET all Rules:
   Curl Request:-

```
curl --location --request GET 'http://localhost:8080/rateLimiterRule
```

3. Scheduler Configuration:

Scheduler will be started at the startup of application with the default values of interval as “120000” miliseconds and intial delay of “3000” miliseconds.

You can configure the both values in application.properties file

```
rateLimiterRuleCache.interval=300000
rateLimiterRuleCache.startupDelay=3000
```

4. Addition of New APIs:

Create a new API in the controller and add it’s signature in ApisIdentifiers:

eg: For Getting all developers, following is the signature-:

```
API_GET_DEVELOPER_ALL(APIConstants.API_METHOD_GET+
UtilityConstants.HYPHEN+APIConstants.API_BASE+APIConstants.API_DEVELOPER)
```

Similarly a rule can be created for specfic user using (1)

5. Implementation of RateLimiterCache:

There are two implementations for RateLimiterCache, One is InMemory Cache and second is RedisServer Cache.

It can be chosen using the following property; default is “InMemory”

```
rateLimiterWindowCacheServiceImpl=RedisServer
```

6. Implementation of Rate Limit Algorithm:

There are two implementations for Rate Limiter Algorithm, One is Fixed Window and second is Sliding Window.

Fixed Window: For Fixed Window counter implementation, a combined key is being generated using userID, API identifier and timestamp (modifed accoding to rate limiter time unit in rule) of the request and counter as value. Whenver a request is landed on Service, counter will be increment and compared with the defined rate for that rule.

Sliding Window(InMemory): For Sliding Window counter implementation, a combined key is generated using userID and API identifier and value is queue of timestamp. Whenver a request is landed on Service, a boudary will be calculated using the rate limiter time unit, and the queue is polled till the boundary is not met. Then it will compare the size of the queue with the rate limit provided in the rule.

Sliding Window(RedisServer): For Sliding Window counter implementation in RedisServer, a combined key of userID and API identifier with a value made of both timestamp with counter is stored in redis. When a request is landed on service, it will store the above explained key:value pair with designated counter and also, set the expiration of that key according to the rate limiter time unit in rule. Then it will compare the no of keys that are present with the rate limit provided in rule.

Note: For Concurrent requests will be handled by concurrent hashmap in InMemory implementationa and by Redis lock (watch) and multi transactions in Redis Server implementation.

You can configure the Algo with the following property; default is `rateLimiterAlgoFixedWindow`

```
rateLimiterAlgo=rateLimiterAlgoSlidingWindow
```

Redis Configuration is done as follows:

```
redis.host=localhost

redis.port=6379

redis.connection.pool=10
```

You can also provide bypass test users, which won’t be resrticed on rate limits

```
rateLimitByPassTestPerformanceUsers="user114","user112"
```

Default rate limit is provided as follows; while default rate limiter time unit is Second

```
rateLimiterDefaultRateLimitPerSecond=10
```

Project:

You can import the project as maven project in any of IDEs and then resolve the dependencies.

Then you can run it as Spring boot application by configuring in same ide.

Note: You need to install and setup redis on your machine in case you’re using RedisServer implementation. (Refer to: https://redis.io/topics/quickstart)

Improvements in Future:

1. A Rule panel can be created to create rules through UI.
2. Redis Cluster can be setup and configuration for the same can be updated in code.
3. More Abstaract layer can be provided more Redis operations.
4. Consul can be used to store properties as key:value pairs

Tests: Test_Results.csv is attached with this file.