API RATE LIMITER

TEAM-4

MANSI
MADHUMITA GANGA S
KEERTHANA R

INDEX

API RATE LIMITER

An Application Program Interface (API) is a set of routines, protocols, and tools for building software applications. Basically, an API specifies how software components should interact.
It is a code that governs the access points for the server.

To protect an API from slow performance and DoS attacks, API owners often enforce a limit on the number of requests, or the quantity of data clients can consume. This is called API Rate Limiting.
Rate limiting is the main element of any API product's scalability.

DIFFERENT APPROACHES AVAILABLE

There are several ways APIs can be rate limited. Three of the most popular ways are:

1. Request Queues

One library sets the rate limit at three requests per second and places the rest in a request queue. There are several plug-and-play options available in the market like Android Volley and Amazon Simple Queue Service (ASQS).

2. Throttling

It lets API developers control how their API is used by setting up a temporary state, allowing the API to assess each request. When the throttle is triggered, a user may either be disconnected or simply have their bandwidth reduced.

3.Rate-limiting Algorithms

Algorithms are another great way to create scalable APIs. As with request queue libraries and throttling services, there are many rate-limiting algorithms available, some of which are:

a. Leaky Bucket

It translates requests into a FIFO (First In First Out) format, If the queue is full, then additional requests are discarded.

Pros

- Processes requests at a regular rate.
- Memory efficient for each user given the limited queue size

Cons

- Can cause starvation of recent requests

b. Token Bucket

It converts the packets into tokens (the total capacity of the bucket) so that only those number of packets as it is permissible and is served. If the queue is full, the token is discarded but not the packet.

Pros

- Packets are not discarded.
- There is no overflow of packets

Cons

- The traffic might not be smoothened.

c. Fixed Window

In a fixed window algorithm, a window size of n seconds is used to track the rate. Each incoming request increments the counter for the window. If the counter exceeds a threshold, the request is discarded

Pros

- No starvation
- Use less memory

Cons

- A single burst of traffic that occurs near the boundary of a window can result in twice the rate of requests being processed.
  - For example, if the rate limit is 5 requests/hour. In the time span of 10AM-11AM, if all the requests come in the time span of 10:30AM-11:00AM, and the next time span of 11AM-12PM, all the requests come in the time span of 11:00AM-11:30AM. Then in the time span of 10:30AM-11:30AM there would be 10 requests. Which violates the 5 requests/hour rate limit.

4. Sliding Log

Timestamped logs are tracked and beyond a threshold they're discarded. The sum of logs determines the rate of an incoming request.

Pros

- High accuracy
- No stampede effects

Cons

- High memory footprint. All the request timestamps needs to be maintained for a window time, thus requires lots of memory to handle multiple users or large window times
- High time complexity for removing the older timestamps

5. Sliding Window

This is a hybrid approach that combines the low processing cost of the fixed window algorithm, and the improved boundary conditions of the sliding log. Each fixed window is tracked using a counter. We account for a weighted value of the previous window's request rate based on the current timestamp.

Pros

- Scalability
- No large memory footprint as only the counts are stored
- No starvation
- No stampede effect

Cons

- Works only for not-so-strict look back window times, especially for smaller unit times

SYSTEM REQUIREMENTS

- Eclipse Enterprise Edition
- Jdk version-13.0.2
- Jar executable file http://www.java2s.com/Code/Jar/e/Downloadexecutablejararchetype10jar.htm
- Apache-tomcat-7.0.105-windows-x64 or any latest version

OUR APPROACH

Studying the pros and cons of the various algorithms, we decided to move forward with the **Sliding Window** approach as it is the most optimized and also uses memory efficiently. Scalability is the most sought-after feature in the market today and the relatively small number of data points needed to track per key allows us to scale and distribute across large clusters

Summary of the algorithm:

If the number of requests served on **configuration key** in the last **time_window_sec** seconds is more than **number_of_requests** configured for it then discard, else the request goes through while we update the counter.

- The window is fixed to 1 minute.
- Default rate limit can be set at the time of object creation. This will only be used if a specific user is either anonymous or the rate limit for the user is not annotated.
- Custom annotations: Custom annotations are created for programmers to input the user rate limits for each API making it plug-and-play.
- Internally, 2 hashmaps are used to keep track of which API is being requested by which User. Each API+User combination has a separate hashmap.
- The user authentication based on user limit and requested api, is done using JWT Authentication

DATA STORAGE APPROACH

The alternative considered was to take an input file from the User and create a hashmap from that. However, since the API had to be plug and play, we settled on feeding in the data through annotations.

ALTERNATIVES EXPLORED

Algorithms

Token bucket and leaky bucket were considered as they are accurate and could have fit this problem statement. Sliding Window was chosen due to its scalability and flexibility.

Database

If any data had to be stored, or if a full application along with service, repository and model/entity classes was to be created, PostgreSQL would have been the preferred database for this project.
But, since plug-and-play was an important feature, making custom annotations was preferred as it is not platform or database dependent.
Database calls are also expensive, whether in-memory databases like H2 or Redis or databases that retrieve from secondary storage like postgreSQL

ASSUMPTIONS

An API might have Regular users, VIP users, Premium VIP users. We use Default, User1 and User2 to represent these priorities respectively and assume that the user given falls in one of these categories rather than indicate an individual user.

Since this project was focused on the API Rate Limiter, we assumed the controller class (NameController) had an object of a service class where the task was executed. Consecutively, we assume that the service had an object of a repository which would refer to a database. Here, we have created a controller class for testing and example purposes only.

ADDITIONAL PARAMETERS WHICH MAY IMPROVE THE EFFICIENCY AND ACCURACY

Tiers+user+API combination can be taken as input. 2 relations that are: which user belongs to which tier and which tier+API combination has how many requests per second.

FUTURE IMPROVEMENTS

- Each API could have its own default rate limit which would increase flexibility.
- Make dynamic objects of class in the main Sliding Window class.
- Internally automate the authentication process facilitating only calling name/getName with the login credentials.
- Adding extra security features to prevent malicious attacks.
- Making a GUI for better presentation.