23rd AUGUST 2020

API RATE LIMITER

Team 7
(Dayananda Sagar Institutions)
Shweta Ranjan
Shreyans Sonthalia
Aishanya Kushwaha

Table Of Contents

1.INTRODUCTION

Rate limiting is a strategy for limiting network traffic. It is a process that is used to define the rate at which users can access APIs. It prevents API's from overuse and DOS attacks by limiting the rate at which user can access an API. Rate limiting blocks users, bots, or applications that are over-using or abusing a web property.

2.OUR UNDERSTANDING

Based on our research, there are 4 types of Rate Limiting:

1. User based Rate Limiting
2. Concurrent Rate Limiting
3. Location/ IP based Rate Limiting
4. Server based Rate Limiting

The various types of Algorithms used for Rate Limiting are:

1. Token-Bucket
2. Leaky bucket
3. Fixed Window
4. Sliding Log
5. Sliding Window

All the above algorithms work effectively with single server setup. But in distributed system, it can raise issues like:

1. Race Conditions
2. Inconsistencies in Rate-Limit Data

To solve above stated problems, we can use:

1. Locking system can be used which will lock the incoming request and once it has been served, only then the other requests will be allowed.
2. Local Memory + Sync Service

3. APPROACH

The approach we have used to solve this problem:

3.1 Approach 1

We decided to build a User Based Rate Limiter which can be efficiently used to prevent abuse and limit the number of requests sent by the user to the API.

A Node.js application has been designed with the assistance of Redis which demonstrates a user, who initially gets authenticated, once authenticated ,he/she can access three APIs namely developers, organizations and employees with specified rate-limits provided by the user while signing-up.

A default rate limit has also been set to handle cases when a user doesn't provide the rate limits explicitly. Sign-up page requests the user to enter the following details :

- Name
- Username

- Password
- Developer Rate Limit
- Organization Rate Limit
- Employee Rate Limit
- Drop down option to select the algorithm to implement rate limit.

Once the user has signed up successfully, the user will be redirected to the home page which consists of links of all the three APIs. The user can access the API a specific number of times which was provided by him/her during signup within 60 seconds .In case a user tries to access an API after his allotted limit or if his/her rate limit gets exhausted, they are redirected to an error page presenting an error.

3.1.1 Assumptions

1. Basic Authentication system (Sign-up/Login).
2. 3 Pages - Developers, Organizations and Employees.
3. Default Rate Limit is 15.
4. Time limit is 60 seconds. However, it can be changed in the code.
5. Username is Unique.
6. Selection of algorithm to implement rate limiting.
7. This is not for distributed systems. However, it works efficiently for single server.

For storing User Information:

HASH-KEY	HASH(TYPE OF VALUE)
USERNAME:NAME	String
USERNAME:PASSWORD	String
USERNAME:DEVELOPERS	Integer
USERNAME:ORGANIZATIONS	Integer
USERNAME:EMPLOYEES	Integer

For storing API Key: Key: TokensLeft(count)

3.1.2 Data Storage Format

This application has been implemented using Redis which is an in-memory key-value store. The performance rate of redis is very high ,it is simple and flexible , interaction with data is command based.

3.1.3 Implementation

Token bucket algorithm and **Sliding Window algorithm** has been used to perform rate limiting.

Token Bucket Algorithm:

- In this algorithm, a counter is maintained to indicate the number of tokens for a specific user and a timestamp showing when it was last updated.
- It is based on a concept where a fixed-capacity bucket is used to hold tokens that are added at a fixed rate.
- The bucket is the database where the user's tokens are stored and expiry is specified. The key in the database is the combination of username+API.

- When a request comes from a specific user, the bucket is checked to see if it contains a sufficient number of tokens as required. If it does, the appropriate number of tokens are decremented and the response is served. Otherwise, it is handled differently.
- Tokens are refilled when there is a difference

Sliding Window Algorithm:

- This algorithm involves tracking a time stamped logs for each user's request.
- These logs are usually stored in a hash set or a hash table which are sorted by time.
- Logs with timestamp beyond a threshold are discarded.
- When a new request comes in, sum of logs is calculated to determine the request rate.

The various commands of Redis namely INCR/DECR and EXPIRE are highly used in building this application. For implementing token bucket algorithm, a Redis key gets created for every minute per API key. Key in our application is Username+API Name. The keys get expired after every 60 seconds .This will prevent database from having any junk data.

In case of Sliding window algorithm, the time stamped logs are stored in a hash set which is sorted by time. When a new request comes in, the sum of logs is calculated to find the request rate and if it reaches beyond the threshold value ,those logs are discarded.

3.1.4 Code Snippet

Token Bucket Algorithm :

```
const moment = require("moment");

/**
 * Token-Bucket Algorithm to limit number of requests coming from the user.
 * @param {string} username
 * @param {string} api
 * @param {string} errorPage
 * @param {object} res
 * @param {object} client
 * @returns
 * This function returns an error page if it encounter an error or else it renders the page requested.
 * If the rate-limit has exceeded then it renders the page limit exceeded page.
 */
function TokenBucket(username, api, errorPage, res, client) {
  var key = username + api;
  //If Key has not get expired
  client.exists(key, function (err, reply) {
    if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});
    if (reply == 1) {
      client.get(key, function (err, count) {
        if (count > 1) {
          res.render(api);
          client.decr(key, function (err, count) {
          });
        } else res.render(errorPage);
      });
```

```
    } else {
      client.hgetall(username, function (err, user) {
        if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});
        if (api == "developers") var access_limit = user.developers;
        else if (api == "organizations")
          var access_limit = user.organizations;
        else if (api == "employees") var access_limit = user.employees;
        //Set Rate-Limit for particular page and user
        client.set(key, access_limit);
        //Key will expire after 1 minute
        client.expire(key, 60);

        if (access_limit > 0) res.render(api);
        else res.render(errorPage);
      });
    }
  });
}
```

Sliding Window Algorithm :

```
/**
 * Sliding-Window Algorithm to limit number of requests coming from the user.
 * @param {string} username
 * @param {string} api
 * @param {string} errorPage
 * @param {object} res
 * @param {object} client
 * @returns
 * This function returns an error page if it encounter an error or else it renders the page requested.
 * If the rate-limit has exceeded then it renders the page limit exceeded page.
 */
function SlidingWindow(username, api, errorPage, res, client) {
  var key = username + api;
  client.exists(key, (err, reply) => {
    if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});
    if (reply == 1) {
      client.get(key, (err, redisResponse) => {
        if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});
        let data = JSON.parse(redisResponse);
        let currentTime = moment().unix();
        let lessThanMinuteAgo = moment().subtract(1, "minute").unix();
        let thresHold = 0;
        data.forEach((item) => {

          if( item.requestTime > lessThanMinuteAgo){

            thresHold += item.counter;

          }
        });

        if (thresHold >= data[0].access_limit) {
```

```
      res.render(errorPage);
    } else {
      let isFound = false;
      data.forEach((element) => {
        if (element.requestTime==currentTime) {
          isFound = true;
          element.counter++;
        }
      });

      if (!isFound) {
        client.hgetall(username, function (err, user) {
          if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});

          if (api == "developers")
            var access_limit = user.developers;
          else if (api == "organizations")
            var access_limit = user.organizations;
          else if (api == "employees")
            var access_limit = user.employees;
          //Set Rate-Limit for particular page and user
          const newrequest = {
            requestTime: currentTime,
            counter: 1,
            access_limit: access_limit,
          };

          data.push(newrequest);
          client.set(key, JSON.stringify(data));
        });
      }else{

        client.set(key, JSON.stringify(data));
      }
      res.render(api);
    }
  });
} else {
  client.hgetall(username, function (err, user) {
    if (err) res.render("error", {message: "There is a Database Error. Please Try Again "});
    if (api == "developers")
       var access_limit = user.developers;
    else if (api == "organizations")
      var access_limit = user.organizations;
    else if (api == "employees")
       var access_limit = user.employees;
    let data = [];
    let requestData = {
      requestTime: moment().unix(),
      counter: 1,
      access_limit: access_limit,
    };
    data.push(requestData);
    client.set(key, JSON.stringify(data));
  });
```

```
    res.render(api);
  }
 });
}

module.exports = { TokenBucket, SlidingWindow };
```

3.1.5 Limitations

The worse-case failure situation could be when the Redis server dies between the INCR and the EXPIRE. This can lead to Database error. When restoring data from in-memory replication, the INCR will not be restored since the transaction was not completed.

Due to this, a user might end up having two limiting keys, one that is being currently used and the other which will expire under the same time window. Apart from this, this application is very efficient.

3.2 Approach 2

In this approach, We can divide the project into two layers, namely, Database layer and User layer.

- Database layer: In this layer, we can implement the front-end using word press and store the user details like username and password in the in-built data base provided by wordpress.
- We can also assign the user roles like organization, employee or developer as used in our project through wordpress.

- User layer: In this layer, we can implement the back-end using redis and python. We can establish the server through redis and implement the API rate limit function using python. Redis command INCR can be used. This will act as a counter. The name of the key should be unique per 1 second so the key name should be rate-limit:<timestamp>. To automatically remove the key from Redis ,we set the timeout of the key for 1 second by checking using TTL and by setting the timeout using EXPIRE. This approach ensures that only one user can login through one IP address by implanting IP snatching and after the limit is exceeded by the user for a certain API , The user can revert back to the screen which was being used by him/her by reloading the blank screen after the time set is crossed.

4. IMPROVEMENTS

We can implement the token bucket algorithm for API rate limiting that satisfies both maximum and minimum rate constraints wherein we are trying to provide both best effort service as well as quality of service of the product.

5. REFERENCES

- https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm/
- https://redislabs.com/redis-best-practices/basic-rate-limiting/
- https://stripe.com/blog/rate-limiters
- https://medium.com/smyte/rate-limiter-df3408325846
- https://www.w3schools.com/nodejs/