**Technical design of Rate limiter**

All API calls are proxied by the Gateway. Checks ratelimit configuration based on user+api. Stores the call counters in Redis. Routes the call only if it hasn’t exceeded the quota, otherwise return error from the gateway itself.

Service1

LB

RateLimiter

(API Gateway)

Service2

smmsm

S

Service1

Service2

RateLimiter

(API Gateway)

Client

LB

Configuration server backed by Git

Redis

All API calls are intercepted by Gateway first, checks the configuration for the rate limit, if the count is less than the limit, call is forwarded to the relevant underlying service, otherwise rejected by Gateway with an error message.

**Why Gateway design?:**

Doing this in gateway allows us to add any API (tech agnostic) and any user without touching the code of Gateway. With the help of Configuration service and cloudbus, we can even achieve this without restarting any component!

I have tried to present this idea by using Git backed config server and actuator/refresh mechanism. But with cloudbus, it will be automatically refreshed without any need of /actuator/refresh API call.

**How configuration is stored:**

There are two tables USER\_API\_LIMIT and API\_INFO. First table stores the rate limit for user+api combination, the second table stores the API related information.

Schema of API\_INFO:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [ID](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [URL](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [RATELIMIT](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [RATE\_LIMIT\_STRATEGY](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [ACCURACY\_LEVEL](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) |
| 1 | /api/v1/books/available | 3 | tunable\_sliding\_window | 3 |

Here, RATELIMIT is the maximum allowed requests/min.

RATE\_LIMIT\_STRATEGY can take two values: BETTER\_FIXED\_WINDOW and TUNABLE\_SLIDING\_WINDOW.

ACCURACY\_LEVEL is applicable only for TUNABLE\_SLIDING\_WINDOW.

Schema of USER\_API\_LIMIT:

|  |  |  |
| --- | --- | --- |
| [USER\_ID](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [API\_ID](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) | [RATELIMIT](http://localhost:8910/h2-console/query.do?jsessionid=a5903d166a3a65a9122463da709b25e8) |
| user1 | 1 | 5 |
| user1 | 3 | 5 |
| user2 | 2 | 15 |

Ratelimit configuration will be first picked from the USER\_API\_LIMIT table, if the record doesn’t exist, it will refer API\_INFO and finally global\_default (default.rate.limit) parameter.

USER+API specific 🡪 API specific default for all users 🡪 Global default for all APIs and all Users

There are admin APIs exposed from the Gateway to change this information dynamically without restart of system. All this is explained in README.md

**Rate limiting algorithms:**

Lot of existing algorithms like leaky bucket, fixed window, sliding low, sliding window are there in market. All of those have their pros and cons.

|  |  |  |
| --- | --- | --- |
| **Name of algo** | **Accuracy** | **Cost complexity** |
| Leaky bucket, fixed window | LOW | LOW |
| Sliding log | HIGHEST | Very high |
| Sliding window | VERY GOOD | Medium |

Out of all the algorithms, sliding window is the most hybrid approach.

**What’s new in this solution**:

Better fixed window:

Rather than naïve implementation of fixed window, have tried to guess the number of requests from the previous window (pro rata basis). Added this extrapolated count to the existing window count to give a fairly accurate count.

This way it gives good accuracy not as high as Sliding window but considerably close. Cost complexity is much lower than sliding window.

TWO GET calls and ONE INCR call to Redis.

Tunable sliding window:

Instead of dividing the window by fixed number of parts. I have made this configurable with ‘accuracyLevel’ parameter. So one can tune the cost complexity to his need. There are three levels:

3 --- Divides the window by 60 parts

2 --- Divides the window by 30 parts

1 --- Divides the window by 20 parts

**Future improvements**:

Currently, /actuator/refresh is used to update the ZuulProperties bean dynamically. This is not needed once cloudbus is activated on config-server.

Right now, ratelimiter-dev.properties need to be updated manually to add new routes. This can be changed in future too by writing a script to change the file in git.

Currently, it loads all the configuration from DB in its local memory on start-up. Assuming the config is not too big, unless we have more than 10million records in USER\_API\_LIMIT table. In case, its bigger than that, we will have to cache this config in Redis or memcache.