[Rate Limiting]

↳ Limiting No. of requests, why

↳ To prevent abuse

E.g → Some wrote infinite loop which fetch from some server

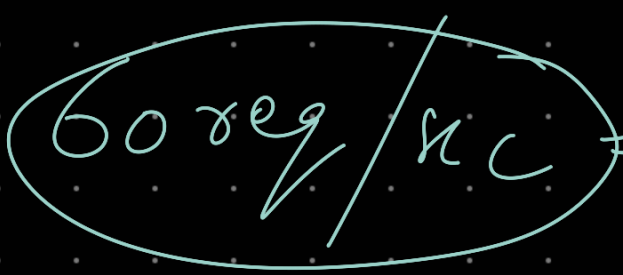
⇓

This can make server crash

What if API endpoint is open?

↳ It can be easily abused

Let's assume 1 client makes

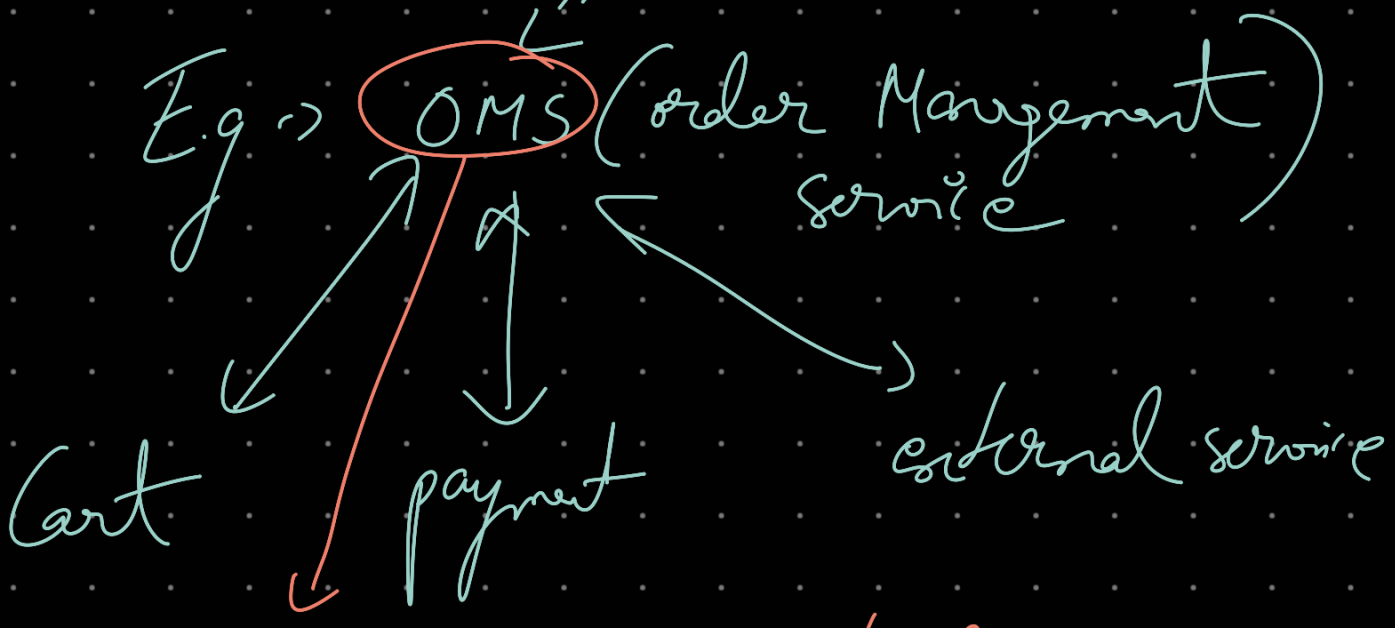


⟹ need to limit

⇓

When to Rate limit?

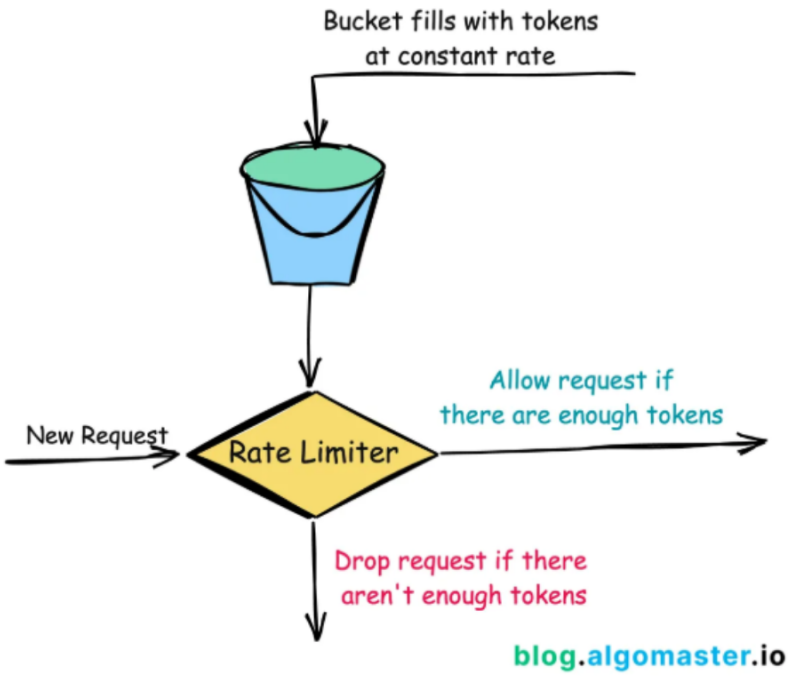
- Open end points
- prevent DDOS for open API
- prevent system failure
- If it's a critical service



OMS could be a critical service in E-commerce.

E.g -> Sentry -> Implements rate limit and drops event
- Cost saving

1. Token Bucket



The Token Bucket algorithm is one of the most popular and widely used rate limiting approaches due to its simplicity and effectiveness.

How It Works:

- Imagine a bucket that holds tokens.
- The bucket has a maximum capacity of tokens.
- Tokens are added to the bucket at a fixed rate (e.g., 10 tokens per second).
- When a request arrives, it must obtain a token from the bucket to proceed.
- If there are enough tokens, the request is allowed and tokens are removed.
- If there aren't enough tokens, the request is dropped.

Pros:

- Relatively straightforward to implement and understand.
- Allows bursts of requests up to the bucket's capacity, accommodating short-term spikes.

Cons:

- The memory usage scales with the number of users if implemented per-user.
- It doesn't guarantee a perfectly smooth rate of requests.

2. Leaky Bucket

The Leaky Bucket algorithm is similar to Token Bucket but focuses on smoothing out bursty traffic.

How it works:

1. Imagine a bucket with a small hole in the bottom.
2. Requests enter the bucket from the top.
3. The bucket processes ("leaks") requests at a constant rate through the hole.
4. If the bucket is full, new requests are discarded.

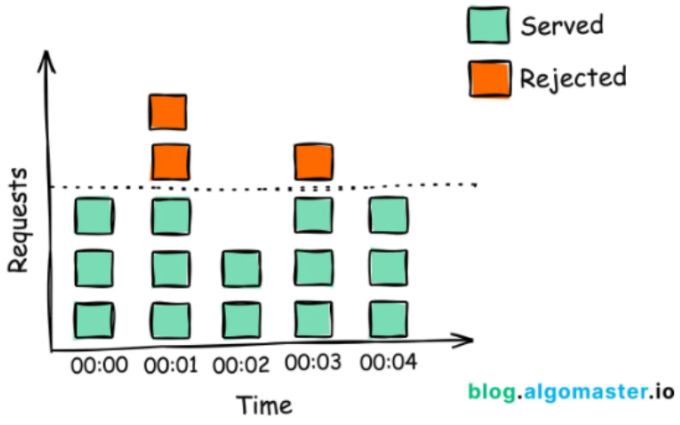
Pros:

- Processes requests at a steady rate, preventing sudden bursts from overwhelming the system.
- Provides a consistent and predictable rate of processing requests.

Cons:

- Does not handle sudden bursts of requests well; excess requests are immediately dropped.
- Slightly more complex to implement compared to Token Bucket.

3. Fixed Window Counter



The Fixed Window Counter algorithm divides time into fixed windows and counts requests in each window.

How it works:

1. Time is divided into fixed windows (e.g., 1-minute intervals).
2. Each window has a counter that starts at zero.
3. New requests increment the counter for the current window.
4. If the counter exceeds the limit, requests are denied until the next window.

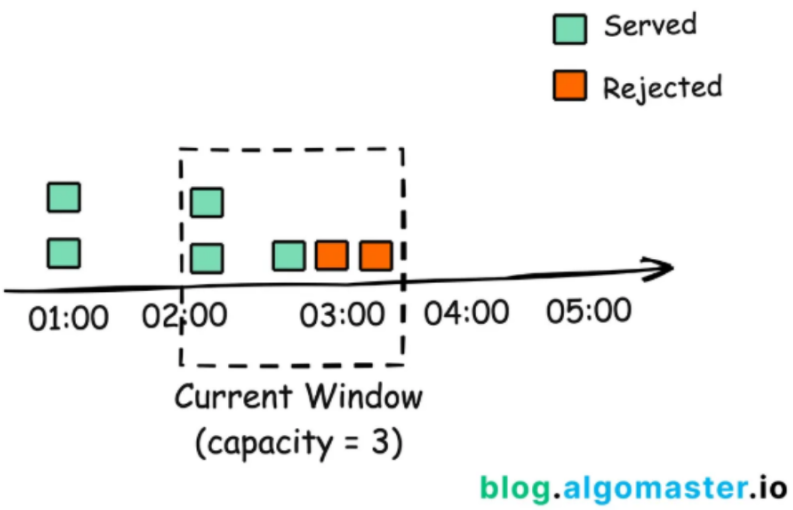
Pros:

- Easy to implement and understand.
- Provides clear and easy-to-understand rate limits for each time window.

Cons:

- Does not handle bursts of requests at the boundary of windows well. Can allow twice the rate of requests at the edges of windows.

4. Sliding Window Log



The Sliding Window Log algorithm keeps a log of timestamps for each request and uses this to determine if a new request should be allowed.

How it works:

1. Keep a log of request timestamps.
2. When a new request comes in, remove all entries older than the window size.
3. Count the remaining entries.
4. If the count is less than the limit, allow the request and add its timestamp to the log.
5. If the count exceeds the limit, request is denied.

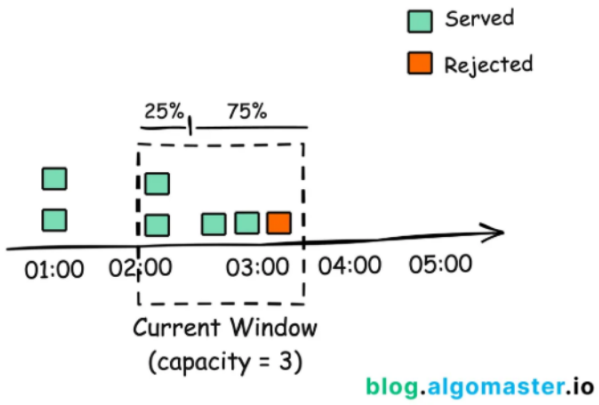
Pros:

- Very accurate, no rough edges between windows.
- Works well for low-volume APIs.

Cons:

- Can be memory-intensive for high-volume APIs.
- Requires storing and searching through timestamps.

5. Sliding Window Counter



This algorithm combines the Fixed Window Counter and Sliding Window Log approaches for a more accurate and efficient solution.

Instead of keeping track of every single request's timestamp as the sliding log does, it focus on the number of requests from the last window.

So, if you are in 75% of the current window, 25% of the weight would come from the previous window, and the rest from the current one:

```
weight = (100 - 75)% * lastWindowRequests + currentWindowRequests
```

Now, when a new request comes, you add one to that weight (weight + 1). If this new total crosses our set limit, we have to reject the request.

How it works:

1. Keep track of request count for the current and previous window.
2. Calculate the weighted sum of requests based on the overlap with the sliding window.
3. If the weighted sum is less than the limit, allow the request.

Pros:

- More accurate than Fixed Window Counter.
- More memory-efficient than Sliding Window Log.
- Smooths out edges between windows.

Cons:

- Slightly more complex to implement.

When implementing rate limiting, consider factors such as the scale of your system, the nature of your traffic patterns, and the granularity of control you need.

Lastly, always communicate your rate limits clearly to your API users, preferably through response headers, so they can implement appropriate retry and backoff strategies in their clients.