API Rate Limits

4.6.1

What are Rate Limits?

REST APIs make it possible to build complex interactions.

Using an API rate limit is a way for a web service to control the number of requests a user or application can make per defined unit of time. Implementing rate limits is best practice for public and unrestricted APIs. Rate limiting helps:

- avoid a server overload from too many requests at once
- provide better service and response time to all users
- protect against a denial-of-service (DoS) attack

Consumers of the API should understand the different algorithm implementations for rate limiting to avoid hitting the limits, but applications should also gracefully handle situations in which those limits are exceeded.

4.6.2

Rate Limit Algorithms

There isn't a standard way to implement rate limiting, but common algorithms include:

- Leaky bucket
- Token bucket
- Fixed window counter
- Sliding window counter

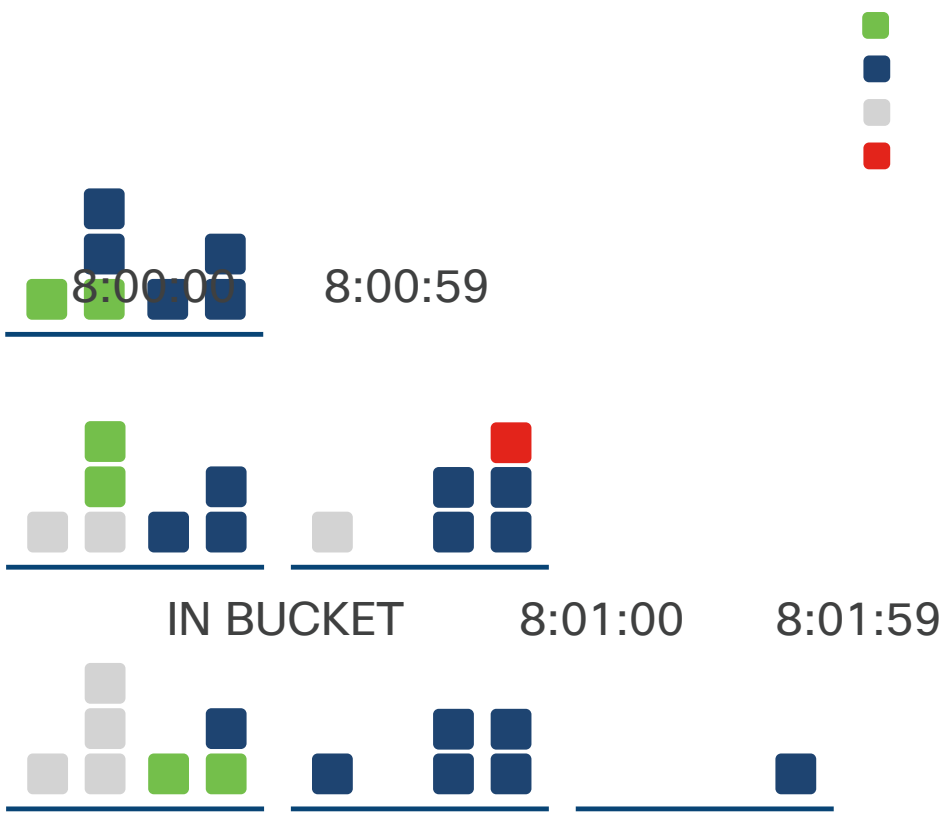
Leaky bucket

Visual representation of the leaky bucket algorithm

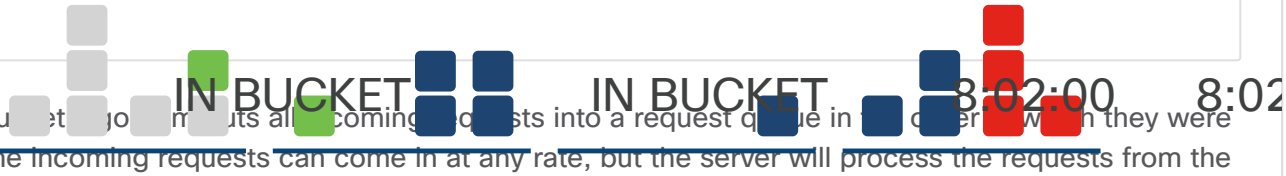
Water cor
inconsiste

ip
r

Rate: 2 Requests / Min
Capacity: 8 Requests



Example of the leaky bucket algorithm

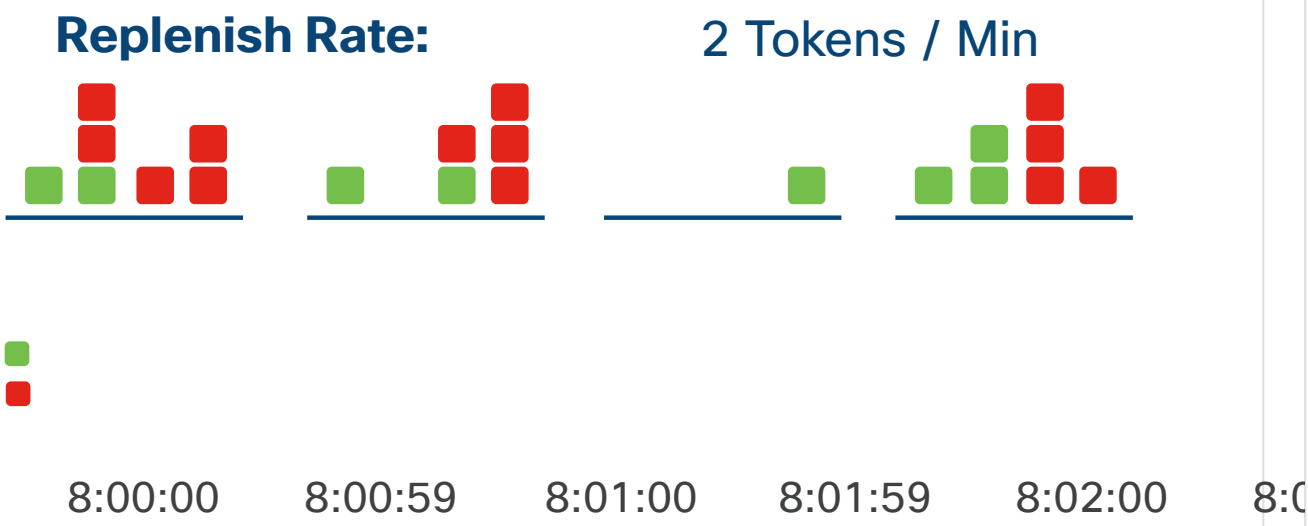


The leaky bucket algorithm puts all incoming requests into a request queue in the order in which they were received. The incoming requests can come in at any rate, but the server will process the requests from the queue at a fixed rate. If the request queue is full, the request is rejected.

With this algorithm, the client must be prepared for delayed responses or rejected requests.

Token bucket

Token bucket algorithm model



Example of the token bucket algorithm

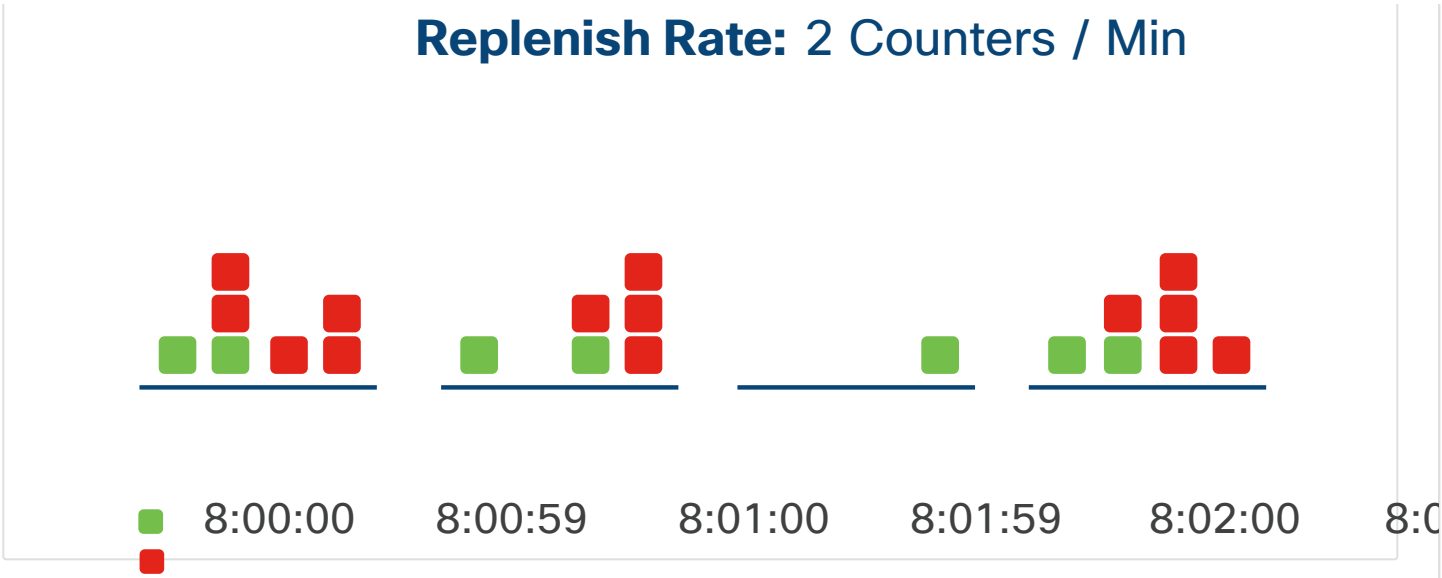
The token bucket algorithm gives each user a defined number of tokens they can use within a certain increment of time, and those tokens accumulate until they're used. When the client does make a request, the server checks the bucket to make sure that it contains at least one token. If so, it removes that token and processes the request. If there isn't a token available, it rejects the request.

All requests made before token replenishment will be rejected. After the tokens are replenished, the user can make requests again.

For example, an API using the Token Bucket algorithm sets a rate limit of 10 requests per client per hour. If a client makes 11 requests within an hour, the 11th request will be rejected because there are no tokens left. On the other hand, if the client then makes no requests for 6 hours, it can then make 60 requests at once, because those tokens have accumulated.

With this algorithm, the client must calculate how many tokens it currently has in order to avoid rejected requests. It also needs to handle the potential rejected requests by building in a retry mechanism for when the tokens are replenished.

Fixed window counter



The fixed window counter algorithm is similar to the token bucket, except for two major differences:

- It uses a counter rather than a collection of tokens.
- The counter cannot be accumulated.

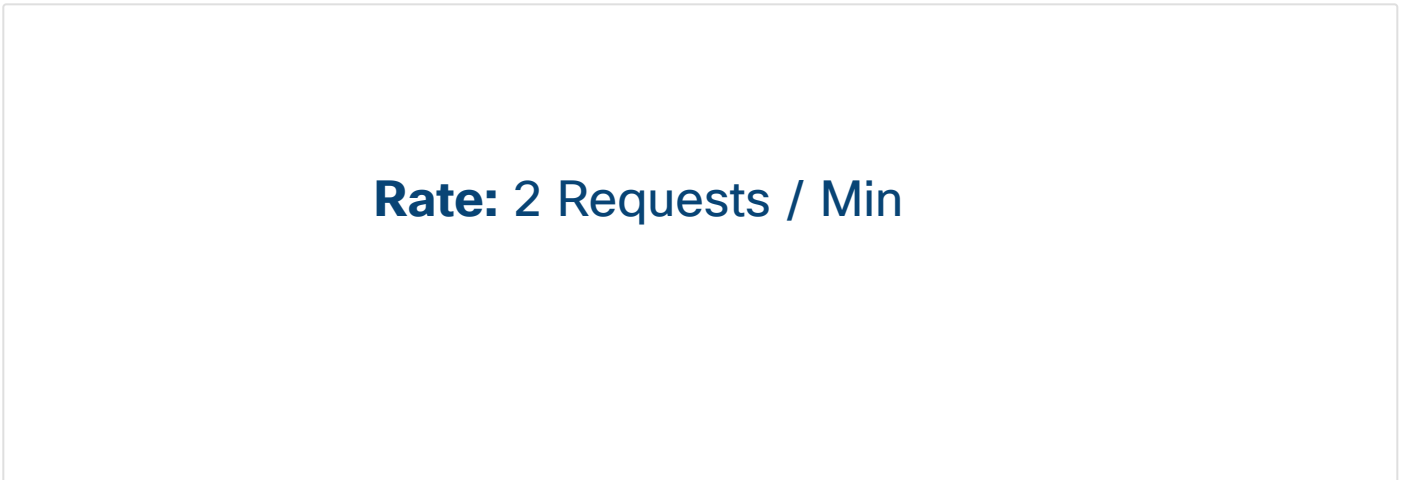
Request OK
Request Rejected

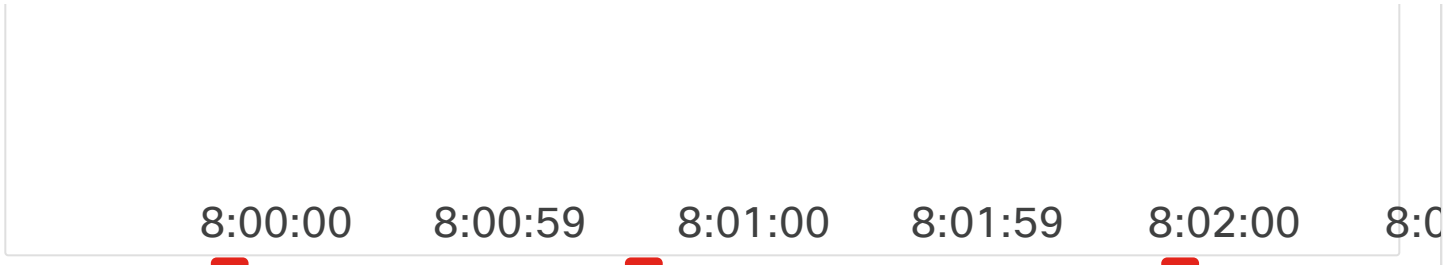
For this algorithm, a fixed window of time is assigned a counter to represent how many requests can be processed during that period. When the server receives a request, the counter for the current window of time is checked to make sure it is not zero. When the request is processed, the counter is deducted. If the limit for that window of time is met, all subsequent requests within that window of time will be rejected. When the next window of time begins, the counter will be set back to the pre-determined count and requests can be processed again.

To go back to our previous example of 10 requests per hour using this algorithm, the 11th request in an hour will still be rejected, but after 6 hours with no requests, the client can still only make 10 requests in a single hour because those "unused" requests were not accumulated.

With this algorithm, the client must know when the window of time starts and ends so that it knows how many requests can be made within that duration of time. Just like the token bucket algorithm, the client must build a retry mechanism so that it can retry the requests when the next window of time has started.

Sliding window counter





The sliding window counter algorithm allows a fixed number of requests to be made in a duration of time. This differs from a fixed window in that the counter is not replenished when the window begins again. In this algorithm, the server stores the timestamp when the request is made. When a new request is made, the server counts how many requests have already been made from the beginning of the window to the current time in order to determine if the request should be processed or rejected. For example, if the rate is five requests per minute, when the server receives a new request, it checks how many requests have been made in the last 60 seconds. If five requests have already been made, then the new request will be rejected.

Request OK

Request Rejected

With this algorithm, the client does not need to know when the window of time starts and ends. It just needs to make sure that the rate limit has not been exceeded at the time of the request. The client also needs to design a way to delay requests if necessary so that it doesn't exceed the allowed rate, and, of course, accommodate rejected requests.

4.6.3

Knowing the Rate Limit

An API's documentation usually provides details of the rate limit and unit of time. In addition, many rate limiting APIs add details about the rate limit in the response's header. Because there isn't a standard, the key-value pair used in the header may differ between APIs. Some commonly used keys include:

- **X-RateLimit-Limit**: The maximum number of requests that can be made in a specified unit of time
- **X-RateLimit-Remaining**: The number of requests remaining that the requester can make in the current rate limit window
- **X-RateLimit-Reset**: The time the rate limit window will reset

The client can use this information to keep track of how many more API requests they can make in the current window, helping the client avoid hitting the rate limit.

4.6.4

Exceeding the Rate Limit

When the rate limit has been exceeded, the server automatically rejects the request and sends back an HTTP response informing the user. In addition, it is common for the response containing the "rate limit exceeded" error to also include a meaningful HTTP status code. Unfortunately, because there isn't a standard for this interaction, the server can choose which status code to send. The most commonly used HTTP status codes are `429: Too Many Requests` or `403: Forbidden`; make sure your client is coded for the specific API it is using.

4.5
Authenticating to a REST API

4.7
Working with Webhooks