# Rate limiting

Ashish Pratap Singh

4 min read



Master System Design
Progress 20/130 chapters
Search topics...
Networking 3/9
API Fundamentals 4/10
What is an API?
Data Formats
API Architectural Styles
WebSockets
Webhooks
WebRTC
API gateways
Rate limiting

Rate limiting helps protects services from being over-whelmed by too many requests from a single user or client.

In this article we will dive into 5 of the most common rate limiting algorithms, their pros and cons and learn how to implement them in code.

Reading Progress 0%
On this page
1. Token Bucket
2. Leaky Bucket
3. Fixed Window Counter
4. Sliding Window Log
5. Sliding Window Counter

## 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.

Idempotency
WebSocket Use Cases
Databases & Storage 0/12
Database Scaling Techniques 0/8
Caching 0/6
Asynchronous Communications 0/4
Tradeoffs 0/9
Distributed System Concepts 0/10
Architectural Patterns 0/5
Microservices 0/6

- 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.

### Code Implementation:

Python Java

```python
import time

class TokenBucket:
    def __init__(self, capacity, fill_rate):
        self.capacity = capacity  # Maximum number c
        self.fill_rate = fill_rate  # Rate at which
        self.tokens = capacity  # Current token cour
        self.last_time = time.time()  # Last time we

    def allow_request(self, tokens=1):
        now = time.time()
        # Calculate how many tokens have been added
        time_passed = now - self.last_time
        self.tokens = min(self.capacity, self.tokens
        self.last_time = now
```

```
        # Check if we have enough tokens for this re
        if self.tokens >= tokens:
            self.tokens -= tokens
            return True
        return False

# Usage example
limiter = TokenBucket(capacity=10, fill_rate=1)  # 1

for _ in range(15):
    print(limiter.allow_request())  # Will print Tru
    time.sleep(0.1)  # Wait a bit between requests

time.sleep(5)  # Wait for bucket to refill
print(limiter.allow_request())  # True
```

**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.

**Code Implementation:**

Python Java

```python
from collections import deque
import time

class LeakyBucket:
    def __init__(self, capacity, leak_rate):
        self.capacity = capacity  # Maximum number c
        self.leak_rate = leak_rate  # Rate at which
        self.bucket = deque()  # Queue to hold reque
        self.last_leak = time.time()  # Last time we

    def allow_request(self):
```

```
        now = time.time()
        # Simulate leaking from the bucket
        leak_time = now - self.last_leak
        leaked = int(leak_time * self.leak_rate)
        if leaked > 0:
            # Remove the leaked requests from the bu
            for _ in range(min(leaked, len(self.buck
                self.bucket.popleft()
            self.last_leak = now

        # Check if there's capacity and add the new
        if len(self.bucket) < self.capacity:
            self.bucket.append(now)
            return True
        return False

# Usage example
limiter = LeakyBucket(capacity=5, leak_rate=1)  # 5

for _ in range(10):
    print(limiter.allow_request())  # Will print Tru
    time.sleep(0.1)  # Wait a bit between requests

time.sleep(1)  # Wait for bucket to leak
print(limiter.allow_request())  # True
```

**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.

**Code Implementation:**

Python Java

```python
import time

class FixedWindowCounter:
    def __init__(self, window_size, max_requests):
        self.window_size = window_size  # Size of th
        self.max_requests = max_requests  # Maximum
        self.current_window = time.time() // window_
        self.request_count = 0

    def allow_request(self):
        current_time = time.time()
        window = current_time // self.window_size

        # If we've moved to a new window, reset the
        if window != self.current_window:
            self.current_window = window
            self.request_count = 0

        # Check if we're still within the limit for
        if self.request_count < self.max_requests:
            self.request_count += 1
            return True
        return False

# Usage example
limiter = FixedWindowCounter(window_size=60, max_rec

for _ in range(10):
    print(limiter.allow_request())  # Will print Tru
```

```
    time.sleep(0.1)  # Wait a bit between requests

time.sleep(60)  # Wait for the window to reset
print(limiter.allow_request())  # True
```

**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 time-stamps 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.

**Code Implementation:**

Python Java

```python
import time
from collections import deque

class SlidingWindowLog:
    def __init__(self, window_size, max_requests):
        self.window_size = window_size  # Size of th
        self.max_requests = max_requests  # Maximum
        self.request_log = deque()  # Log to keep tr

    def allow_request(self):
        now = time.time()

        # Remove timestamps that are outside the cur
        while self.request_log and now - self.reques
            self.request_log.popleft()

        # Check if we're still within the limit
        if len(self.request_log) < self.max_requests
            self.request_log.append(now)
```

```
            return True
        return False

# Usage example
limiter = SlidingWindowLog(window_size=60, max_reque

for _ in range(10):
    print(limiter.allow_request())  # Will print Tru
    time.sleep(0.1)  # Wait a bit between requests

time.sleep(60)  # Wait for the window to slide
print(limiter.allow_request())  # True
```

**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 time-stamp 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.

**Code Implementation:**

Python Java

```python
import time

class SlidingWindowCounter:
    def __init__(self, window_size, max_requests):
        self.window_size = window_size  # Size of th
        self.max_requests = max_requests  # Maximum
        self.current_window = time.time() // window_
        self.request_count = 0
        self.previous_count = 0

    def allow_request(self):
        now = time.time()
        window = now // self.window_size

        # If we've moved to a new window, update the
        if window != self.current_window:
            self.previous_count = self.request_count
            self.request_count = 0
            self.current_window = window

        # Calculate the weighted request count
        window_elapsed = (now % self.window_size) /
        threshold = self.previous_count * (1 - windo

        # Check if we're within the limit
        if threshold < self.max_requests:
            self.request_count += 1
            return True
        return False
```

```
# Usage example
limiter = SlidingWindowCounter(window_size=60, max_r

for _ in range(10):
    print(limiter.allow_request())  # Will print Tru
    time.sleep(0.1)  # Wait a bit between requests

time.sleep(30)  # Wait for half the window to pass
print(limiter.allow_request())  # Might be True or F
```

**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.

Prev: API gateways
Take Notes
Star
Mark as Complete
Ask AI
New
Next: Idempotency