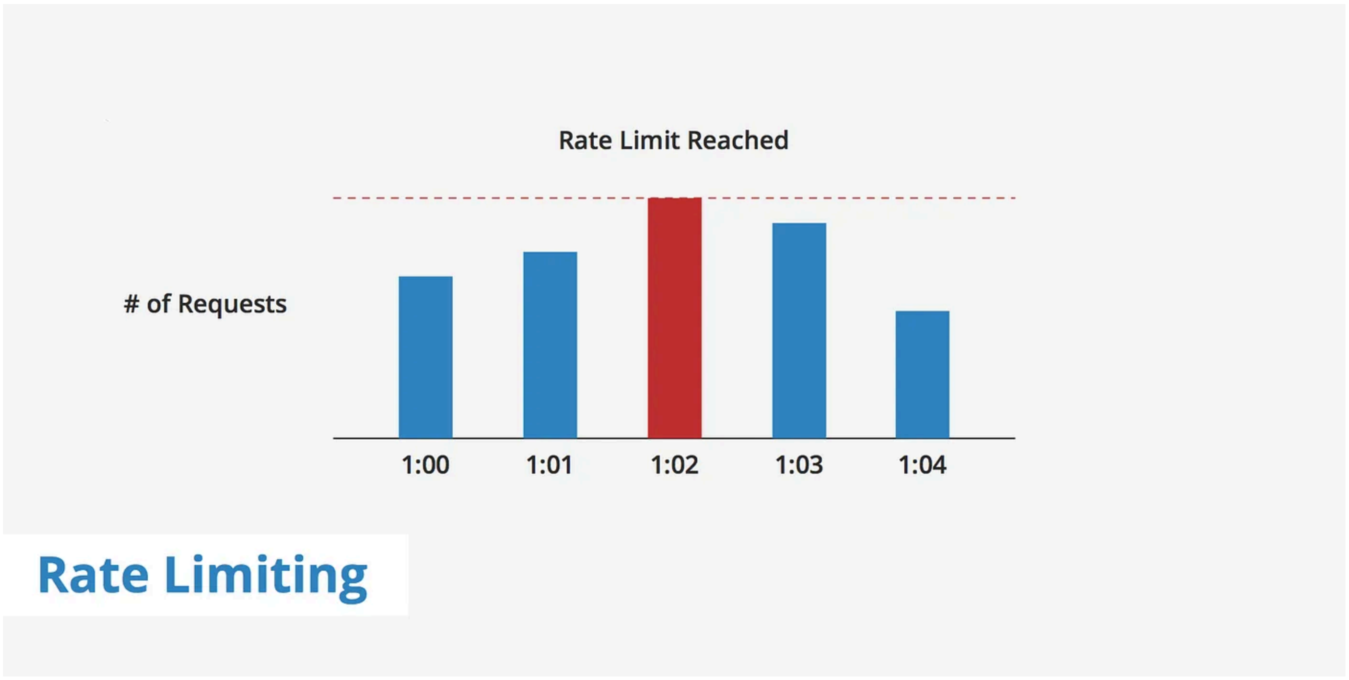
Rate Limiting Algorithms using Redis

Rahul · Follow
6 min read · Aug 28, 2020

Listen Share

Fixed Window, Sliding Logs, Leaky Bucket, Sliding Window, Token Bucket



rate limiting

A very brief introduction of Rate Limiting.

Rate-limiting is a procedure that allows you to control the rate at which your users can send requests to your server. Rate limiting is mostly used to protect your servers from unwanted bursts, malicious attacks.

Rate limiting can be handled both on the server and client-side. On the server-side, rate limiting can be implemented with

1. Proxy servers like Nginx

2. An in-memory high-performance database like Redis
3. Third-party services like AWS API Gateway,
4. Using your data structure and algorithms in your code.

For now, let’s use Redis to implement rate-limiting using some of the most used rate-limiting algorithms and discuss their pros and cons.

The following are the algorithms discussed and implemented.

1. Fixed Window
2. Sliding Logs
3. Leaky Bucket
4. Sliding Window
5. Token Bucket

Fixed Window

In this algorithm, we use fixed window intervals to count requests. For 100 req/min, we use 1 minute/60 second interval windows. Each incoming request increments the counter for the currently active window (windows are defined by the floor of the current timestamp), so with a 1 minute/60 second window, if the current time is 10:00:27, the current window would be 10:00:00 and if time is 10:01:27, the window would be 10:01:00.

For each request a user makes,

1. Check if the user has exceeded the limit in the current window.
2. If the user has exceeded the limit, the request is dropped
3. Otherwise, we increment the counter

```go
// RateLimitUsingFixedWindow .
func RateLimitUsingFixedWindow(userID string, intervalInSeconds int64, maximumRequests
	// userID can be apikey, location, ip
	currentWindow := strconv.FormatInt(time.Now().Unix()/intervalInSeconds, 10)
	key := userID + ":" + currentWindow // user userID + current time window
	// get current window count
	value, _ := redisClient.Get(ctx, key).Result()
	requestCount, _ := strconv.ParseInt(value, 10, 64)
	if requestCount >= maximumRequests {
		// drop request
		return false
	}

	// increment request count by 1
	redisClient.Incr(ctx, key) // if the key is not available, value is initialised

	// handle request
	return true
	// delete all expired keys at regular intervals
}
```

fixed_window.go hosted with ❤ by GitHub view raw

At regular intervals, say every 10 min or every hour, delete all the expired window keys (instead of setting the current window to expire at the start next window using EXPIRE command in Redis, which is another command/load on Redis for every request).

Pros

- Easy to implement.

Cons

- A burst of requests at the end of the window causes server handling more requests than the limit since this algorithm will allow requests for both current and next window requests within a short time. For example, for 100 req/min, if the user makes 100 requests at 55 to 60 seconds window and then 100 requests from 0 to 5 seconds in the next window, this algorithm handles all the requests. Thus, ends up handling 200 requests in 10 seconds for this user, which is above the limit of 100 req/min

Sliding Logs

In this algorithm, we log every request a user makes into a sorted list(sorted with time).

For each request a user makes,

1. Check if the user has exceeded the limit in the current window by getting the count of all the logs in the last window in the sorted set. If current time is 10:00:27 and rate is 100 req/min, get logs from previous window(10:00:27–60= 09:59:28) to current time(10:00:27).

2. If the user has exceeded the limit, the request is dropped.

3. Otherwise, we add the unique request ID (you can get from the request or you can generate a unique hash with userID and time) to the sorted sets(sorted by time).

```go
// RateLimitUsingSlidingLogs .
func RateLimitUsingSlidingLogs(userID string, uniqueRequestID string, intervalInSeconds
	// userID can be apikey, location, ip
	currentTime := strconv.FormatInt(time.Now().Unix(), 10)
	lastWindowTime := strconv.FormatInt(time.Now().Unix()-intervalInSeconds, 10)
	// get current window count
	requestCount := redisClient.ZCount(ctx, userID, lastWindowTime, currentTime).Va
	if requestCount >= maximumRequests {
		// drop request
		return false
	}

	// add request id to last window
	redisClient.ZAdd(ctx, userID, &redis.Z{Score: float64(time.Now().Unix()), Membe

	// handle request
	return true
	// remove all expired request ids at regular intervals using using ZRemRangeByS
}
```

sliding_logs.go hosted with ❤ by GitHub view raw

At regular intervals, say every 10 min or every hour, delete all the expired request IDs from sorted sets using ZRemRangeByScore from -inf to last window time.

Pros

- Overcomes cons of the fixed window by not imposing a fixed window limit and thus unaffected by bursts of requests at the end of the window

Cons

- It is not memory-efficient since we store a new entry for every request made.

- It is very expensive because we count the user's last window requests in each request, which doesn't scale well when large bursts of attacks happen.

Leaky Bucket

In this algorithm, we use limited sized queue and process requests at a constant rate from queue in First-In-First-Out(FIFO) manner.

For each request a user makes,

1. Check if the queue limit is exceeded.
2. If the queue limit has exceeded, the request is dropped.
3. Otherwise, we add requests to queue end and handle the incoming request.

```go
// RateLimitUsingLeakyBucket .
func RateLimitUsingLeakyBucket(userID string, uniqueRequestID string, intervalInSeconds
	// userID can be apikey, location, ip
	requestCount := redisClient.LLen(ctx, userID).Val()
	if requestCount >= maximumRequests {
		// drop request
		return false
	}

	// add request id to the end of queue
	redisClient.RPush(ctx, userID, uniqueRequestID)

	// handle request
	return true
}
```

leaky_bucket.go hosted with ❤ by GitHub view raw

Requests are processed at a constant rate from the queue in a FIFO manner(removed from the start of the queue and handled) from a background process. (you can use LPOP command in Redis at a constant rate, for example for 60 req/min, you can remove 1 element per second and handle the removed request)

Pros

- Overcomes the cons of the fixed window by not imposing a fixed window limit and thus unaffected by a burst of requests at the end of the window.

- Overcomes the cons of sliding logs by not storing all the requests(only the requests limited to queue size) and thus memory efficient.

Cons

- Bursts of requests can fill up the queue with old requests and most recent requests are slowed from being processed and thus gives no guarantee that requests are processed in a fixed amount of time.
- This algorithm causes traffic shaping(handling requests at a constant rate, which prevents server overload, a plus point), which slows user's requests and thus affecting your application.

Sliding Window

In this algorithm, we combine both the fixed window and the sliding log algorithm. We maintain a counter for each fixed window and we account for the weighted counter value of previous window request count along with current window request count.

For each request a user makes,

1. Check if the user has not exceeded the limit in the current window.
2. If the user has exceeded, the request is dropped
3. Otherwise, we calculate the weighted count of the previous window, for example, if the current window time has been elapsed by 30%, then we weight the previous window's count by 70%
4. Check if the previous window weighted count and current window count has exceeded the limit.
5. If the user has exceeded, the request is dropped.
6. Otherwise, we increment the counter by 1 in the current window and handle the incoming request.

```

// RateLimitUsingSlidingWindow .
func RateLimitUsingSlidingWindow(userID string, uniqueRequestID string, intervalInSecon
	// userID can be apikey, location, ip
	now := time.Now().Unix()

	currentWindow := strconv.FormatInt(now/intervalInSeconds, 10)
	key := userID + ":" + currentWindow // user userID + current time window
	// get current window count
	value, _ := redisClient.Get(ctx, key).Result()
	requestCountCurrentWindow, _ := strconv.ParseInt(value, 10, 64)
	if requestCountCurrentWindow >= maximumRequests {
		// drop request
		return false
	}

	lastWindow := strconv.FormatInt(((now - intervalInSeconds) / intervalInSeconds)
	key = userID + ":" + lastWindow // user userID + last time window
	// get last window count
	value, _ = redisClient.Get(ctx, key).Result()
	requestCountlastWindow, _ := strconv.ParseInt(value, 10, 64)

	elapsedTimePercentage := float64(now%intervalInSeconds) / float64(intervalInSec

	// last window weighted count + current window count
	if (float64(requestCountlastWindow)*(1-elapsedTimePercentage))+float64(requestC
		// drop request
		return false
	}

	// increment request count by 1 in current window
	redisClient.Incr(ctx, userID+":"+currentWindow)

	// handle request
	return true
}
```

sliding_window.go hosted with ❤ by GitHub view raw

At regular intervals, say every 10 min or every hour, delete all the expired window keys.

Pros

- Overcomes the cons of the fixed window by not imposing a fixed window limit and thus unaffected by a burst of requests at the end of the window.

- Overcomes the cons of sliding logs by not storing all the requests and avoiding counting for every request and thus memory and performance efficient.
- Overcomes the cons of leaky bucket starvation problem by not slowing requests, not traffic shaping.

Cons

- Not a con, but you need to delete expired window keys, an extra command/load on Redis, which you can overcome in the next algorithm.

Token Bucket

In this algorithm, we maintain a counter which shows how many requests a user has left and time when the counter was reset.

For each request a user makes,

1. Check if window time has been elapsed since the last time counter was reset. For rate 100 req/min, current time 10:00:27, last reset time 9:59:00 is not elapsed and 9:59:25(10:00:27–9:59:25 > 60 sec) is elapsed.
2. If window time is not elapsed, check if the user has sufficient requests left to handle the incoming request.
3. If the user has none left, the request is dropped.
4. Otherwise, we decrement the counter by 1 and handle the incoming request.
5. If the window time has been elapsed, i.e., the difference between last time counter was reset and the current time is greater than the allowed window(60s), we reset the number of requests allowed to allowed limit(100)

```go
// RateLimitUsingTokenBucket .
func RateLimitUsingTokenBucket(userID string, intervalInSeconds int64, maximumRequests
	// userID can be apikey, location, ip
	value, _ := redisClient.Get(ctx, userID+"_last_reset_time").Result()
	lastResetTime, _ := strconv.ParseInt(value, 10, 64)
	// if the key is not available, i.e., this is the first request, lastResetTime
	// check if time window since last counter reset has elapsed
	if time.Now().Unix()-lastResetTime >= intervalInSeconds {
		// if elapsed, reset the counter
		redisClient.Set(ctx, userID+"_counter", strconv.FormatInt(maximumReques
	} else {
		value, _ := redisClient.Get(ctx, userID+"_counter").Result()
		requestLeft, _ := strconv.ParseInt(value, 10, 64)
		if requestLeft <= 0 { // request left is 0 or < 0
			// drop request
			return false
		}
	}

	// decrement request count by 1
	redisClient.Decr(ctx, userID+"_counter")

	// handle request
	return true
}
```

token_bucket.go hosted with ❤️ by GitHub view raw

In this algorithm, there is no need for background code to check and delete expired keys.

Pros

- Overcomes all the above algorithms cons, no fixed window limit, memory and performance efficient, no traffic shaping.
- No need for background code to check and delete expired keys.

Finally, I would say not every algorithm suites all the problems, use whichever is comfortable to your needs. But I prefer and recommend Token Bucket.

That’s all for now. Let me know about your views, comment below for any clarifications

If you like my article, please don't forget to click 👏👏👏 and share.

Also, to be notified about my new articles and stories, follow me on Medium and Twitter. You can find me on LinkedIn as well. Cheers!

Rate Limiting Golang Redis API

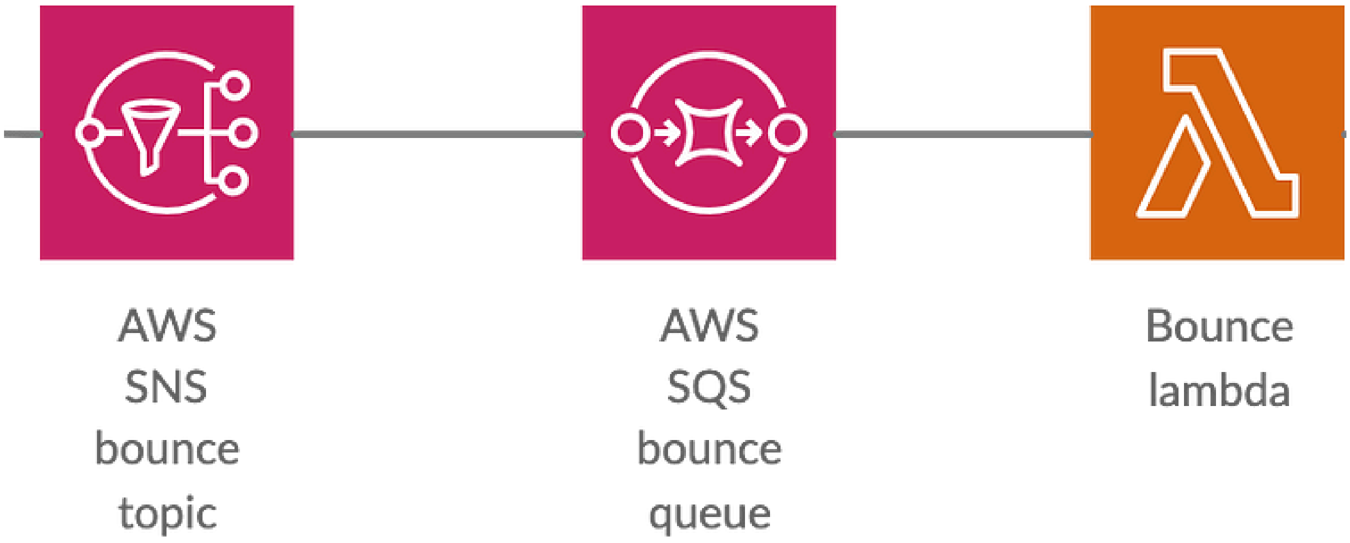
Follow

Written by Rahul

201 Followers

Backend developer. Tech Blogger. Find me @ https://sairahul.me

More from Rahul



Rahul

Handling AWS SES bounces and complaints

Amazon SES recommends bounce rate below 2%. Accounts with bounce rates exceeding 5% will be placed under review and exceeding 10% will be...

Jun 12, 2020 9

Rahul

The Ultimate Guide to Indexing Your Database for Lightning-Fast Queries!

Learn various indexes available in Postgres database.

Jan 12, 2023 35 2

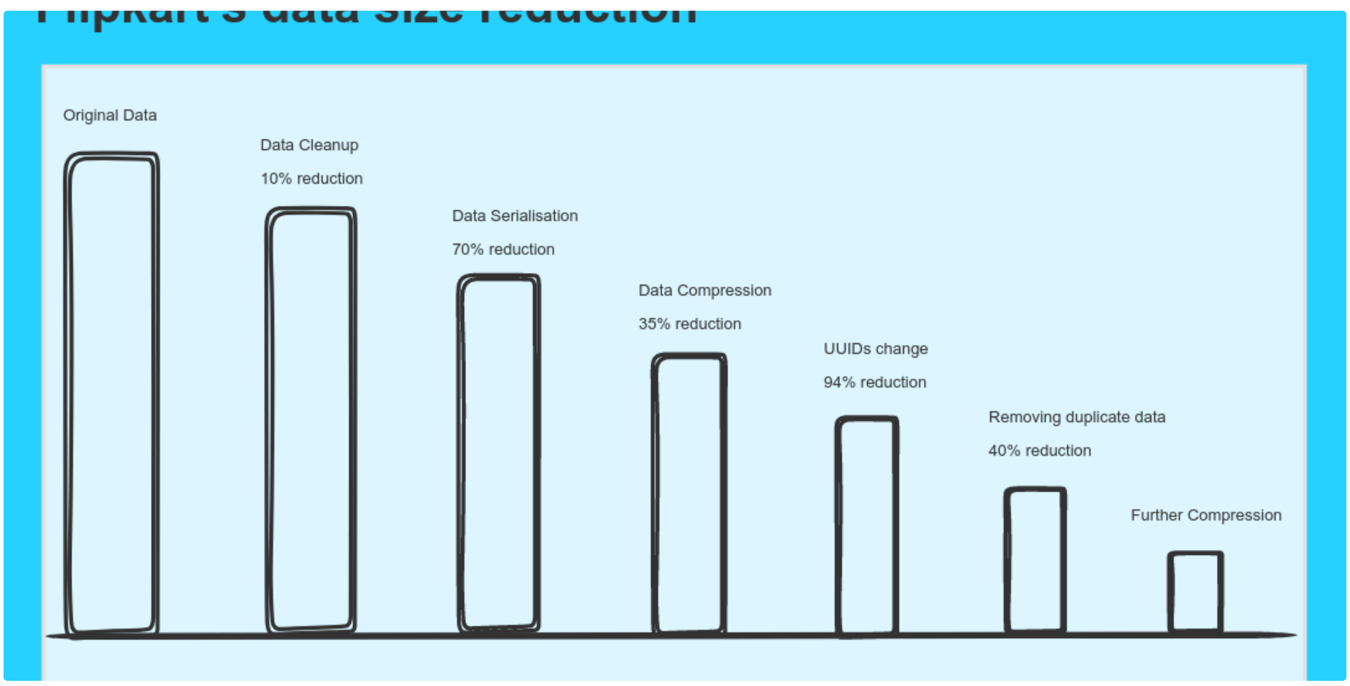
9:41
Monday, June 7
John Baily now
Can you bring a big salad? I'm on dessert duty.
Headspace 2m ago
When it comes to meditation, every day is the first day. Each new moment, the first moment.
Neighborhood takeout favorites 2m ago
Need inspiration? Kea Mao from Up Thai is a popular takeout option in your area.
Hugo Verweij 3m ago
Never a dull moment!

Rahul

Can you send Notifications without internet on user's device?

Learn how Myntra used a very simple trick to send notifications to it's users even without internet on their devices

Nov 25, 2022 1



Rahul

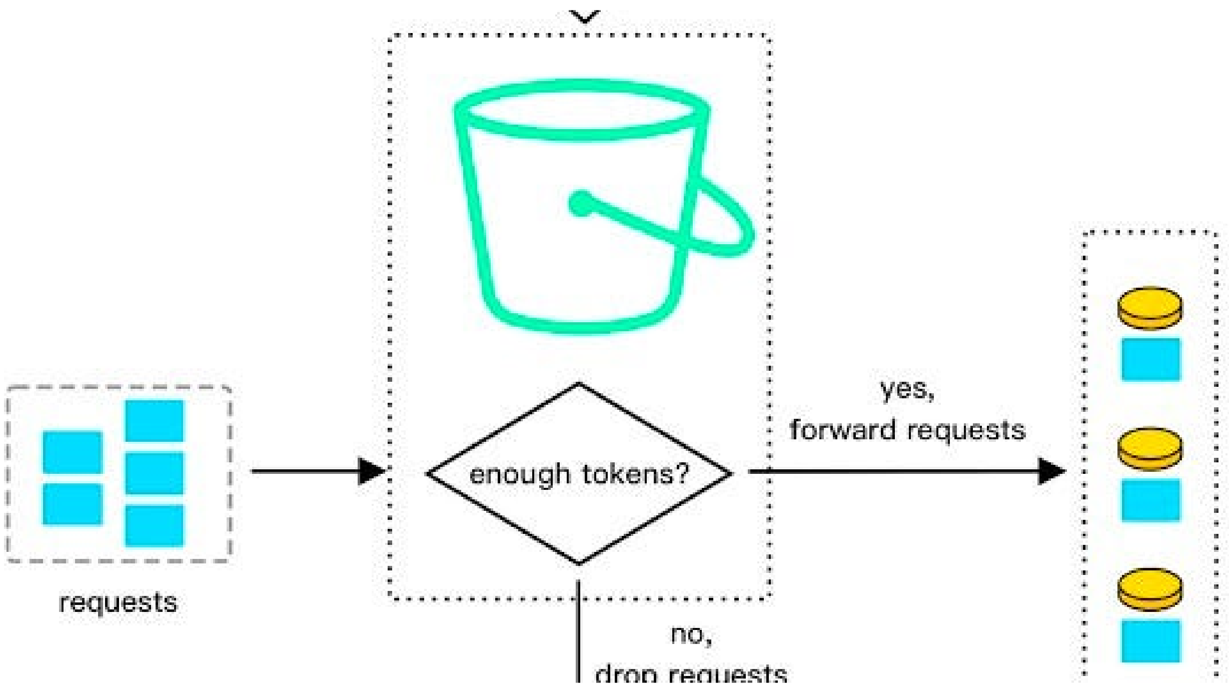
How Flipkart reduced their data size by 96%

Learn how Flipkart decreased their data needs drastically without any change in performance.

Nov 10, 2022 200 8

See all from Rahul

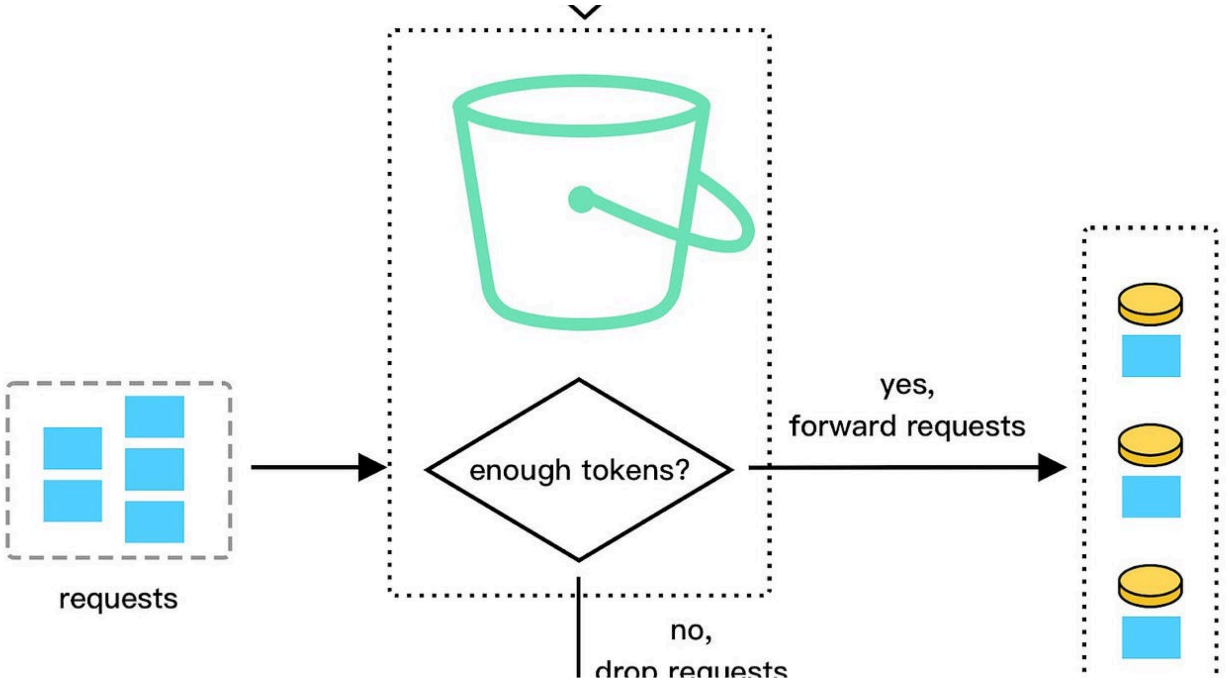
Recommended from Medium



V Vijay

Rate Limiter

May 12 10



David Lee in Towards Dev

Leaky Bucket vs Token Bucket in Rate Limiting Algorithms

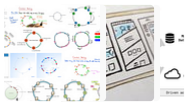
Leaky Bucket and Token Bucket Rate Limiting Algorithms

Feb 22 166

Lists

Coding & Development

11 stories · 743 saves

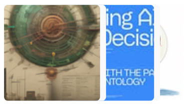


General Coding Knowledge

20 stories · 1483 saves

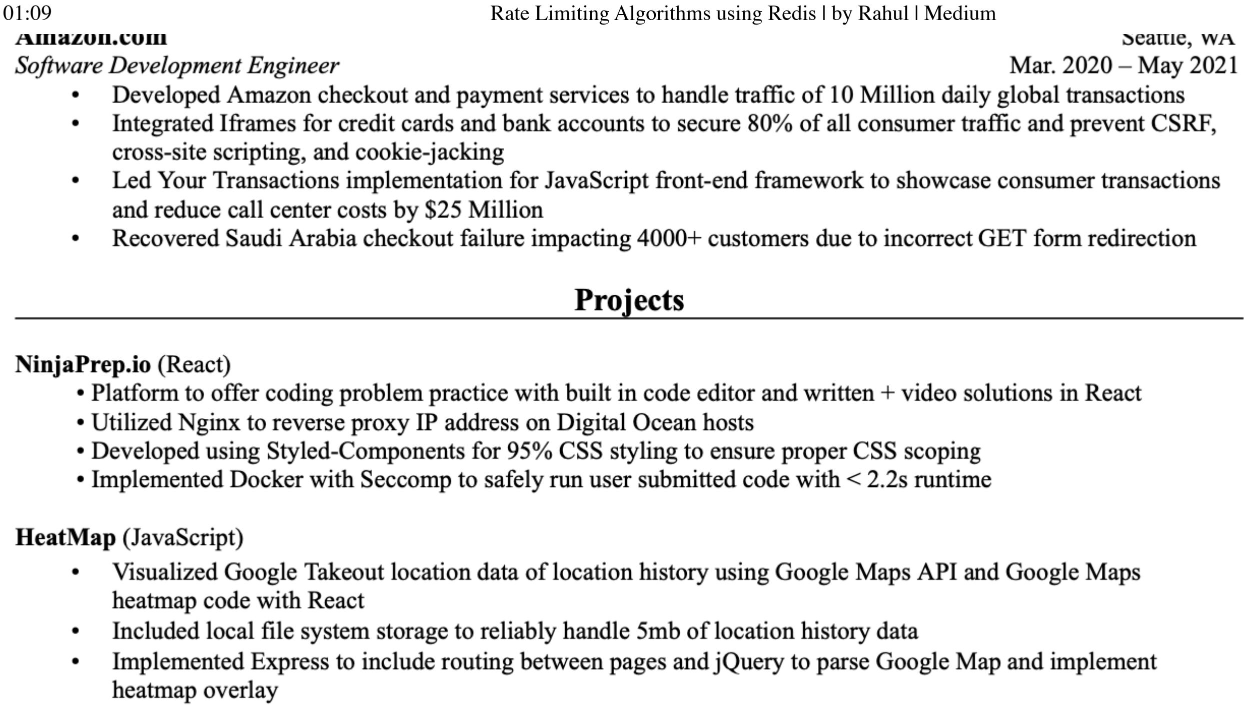
Company Offsite Reading List

8 stories · 138 saves



data science and AI

40 stories · 216 saves



Amazon.com Seattle, WA

Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)

- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)

- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

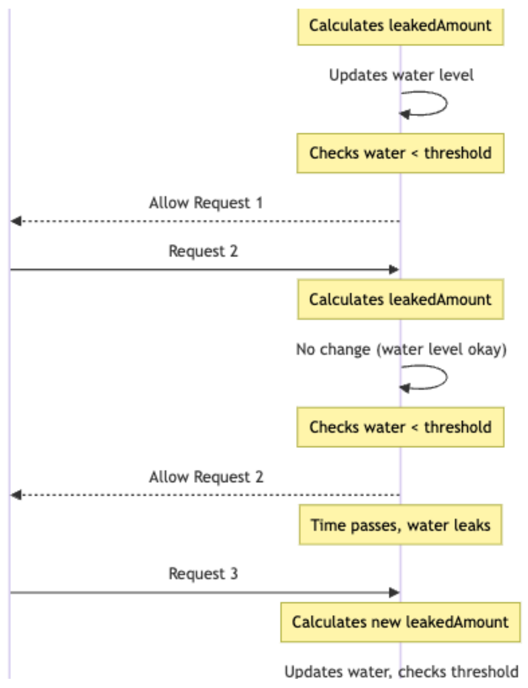
Jun 1 18K 283

Kartik Parihar

Rate Limiting

Ensuring Fair Usage and Quality of Service

Mar 22 2



Deven CHEN

Implementing Rate Limiting in Java from Scratch—Leaky Bucket and Tokenn Bucket implementation

(continue)

Mar 4 8 4

Ayush Nandanwar

Achieving Distributed Locking in Node.js with Redis and Redlock

In a distributed system where multiple processes or instances are running concurrently, ensuring that only one process can access a shared…

Apr 26 50

See more recommendations