Token bucket

Each user has their own bucket of tokens that gets refilled at a set interval. A token is removed on every request until none is left and the request is rejected. While a bit more complex than the fixed-window algorithm, it allows you to handle initial bursts and process requests more smoothly overall.

Memory storage

This requires the server to persist its memory across requests and will not work in serverless environments.

```
export class TokenBucketRateLimit<_Key> {
    public max: number;
    public refillIntervalSeconds: number;

    constructor(max: number, refillIntervalSeconds: number) {
        this.max = max;
        this.refillIntervalSeconds = refillIntervalSeconds;
    }

    private storage = new Map<_Key, Bucket>();

    public consume(key: _Key, cost: number): boolean {
        let bucket = this.storage.get(key) ?? null;
        const now = Date.now();
        if (bucket === null) {
            bucket = {
                count: this.max - cost,
                refilledAt: now
            };
            this.storage.set(key, bucket);
            return true;
        }
        const refill = Math.floor((now - bucket.refilledAt) / (this.refillIntervalS
        bucket.count = Math.min(bucket.count + refill, this.max);
        bucket.refilledAt = bucket.refilledAt + refill * this.refillIntervalSeconds
        if (bucket.count < cost) {
            return false;
        }
        bucket.count -= cost;
        this.storage.set(key, bucket);
        return true;
    }
```

```
}

interface Bucket {
    count: number;
    refilledAt: number;
}
```

```
// Bucket that has 10 tokens max and refills at a rate of 2 tokens/sec
const ratelimit = new TokenBucketRateLimit<string>(10, 2);

if (!ratelimit.consume(ip, 1)) {
    throw new Error("Too many requests");
}
```

Redis

We'll use Lua scripts to ensure queries are atomic.

```
-- Returns 1 if allowed, 0 if not
local key                   = KEYS[1]
local max                   = tonumber(ARGV[1])
local refillIntervalSeconds = tonumber(ARGV[2])
local cost                  = tonumber(ARGV[3])
local now                   = tonumber(ARGV[4]) -- Current unix time in seconds

local fields = redis.call("HGETALL", key)

if #fields == 0 then
    local expiresInSeconds = cost * refillIntervalSeconds
    redis.call("HSET", key, "count", max - cost, "refilled_at", now)
    redis.call("EXPIRE", key, expiresInSeconds)
    return {1}
end

local count = 0
local refilledAt = 0
for i = 1, #fields, 2 do
    if fields[i] == "count" then
        count = tonumber(fields[i+1])
    elseif fields[i] == "refilled_at" then
        refilledAt = tonumber(fields[i+1])
    end
end

local refill = math.floor((now - refilledAt) / refillIntervalSeconds)
count = math.min(count + refill, max)
refilledAt = refilledAt + refill * refillIntervalSeconds
```

```
if count < cost then
    return {0}
end

count = count - cost
local expiresInSeconds = (max - count) * refillIntervalSeconds
redis.call("HSET", key, "count", count, "refilled_at", now)
redis.call("EXPIRE", key, expiresInSeconds)
return {1}
```

Load the script and retrieve the script hash.

```
const SCRIPT_SHA = await client.scriptLoad(script);
```

Reference the script with the hash.

```
export class TokenBucketRateLimit {
    private storageKey: string;

    public max: number;
    public refillIntervalSeconds: number;

    constructor(storageKey: string, max: number, refillIntervalSeconds: number) {
        this.storageKey = storageKey;
        this.max = max;
        this.refillIntervalSeconds = refillIntervalSeconds;
    }

    public async consume(key: string, cost: number): Promise<boolean> {
        const result = await client.EVALSHA(SCRIPT_SHA, {
            keys: [`${this.storageKey}:${key}`],
            arguments: [
                this.max.toString(),
                this.refillIntervalSeconds.toString(),
                cost.toString(),
                Math.floor(Date.now() / 1000).toString()
            ]
        });
        return Boolean(result[0]);
    }
}
```

```
// Bucket that has 10 tokens max and refills at a rate of 2 tokens/sec.
// Ensure that the storage key is unique.
const ratelimit = new TokenBucketRateLimit("global_ip", 10, 2);

const valid = await ratelimit.consume(ip, 1);
```

```
if (!valid) {
    throw new Error("Too many requests");
}
```