Redis Rate Limiter

Production Documentation

Token Bucket Algorithm with Distributed State

Version 1.0.0
Generated: 2025-12-16 22:02
Jules MCP Server - Antigravity Orchestration

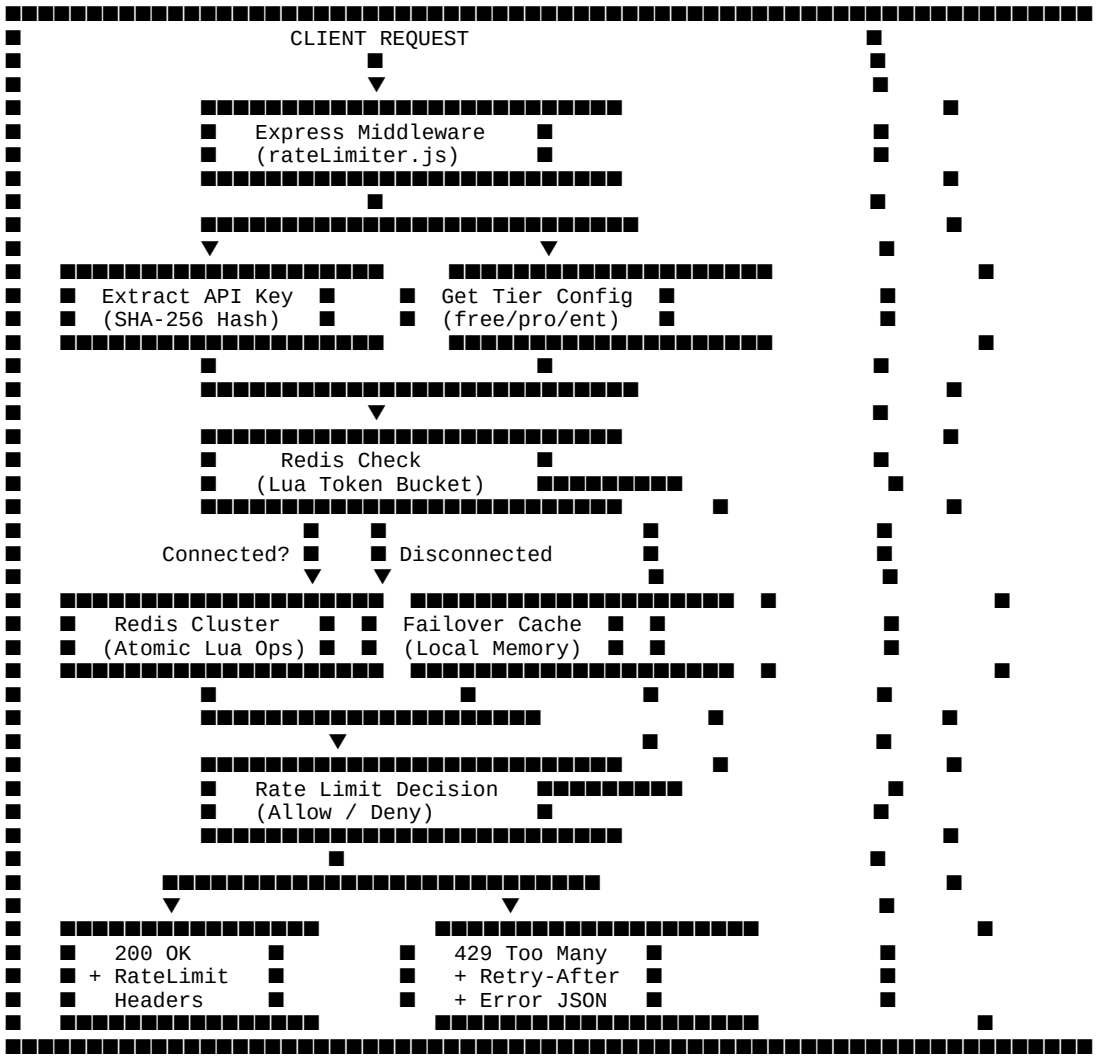
Table of Contents

1. Architecture Overview

System Architecture

The rate limiter implements a distributed token bucket algorithm using Redis for state management. It supports per-API-key rate limiting with tiered configurations and graceful failover to local memory when Redis is unavailable.

Architecture Diagram



2. Tier Configuration

The rate limiter supports three tiers with configurable limits. Each tier uses the token bucket algorithm with different refill rates and burst capacities.

Tier	Requests/Min	Burst Capacity	Refill Rate	Window	Bypass
Free	100	150	1.67/sec	60s	No
Pro	1,000	1,500	16.67/sec	60s	No
Enterprise	10,000	15,000	166.67/sec	60s	Yes

Endpoint-Specific Overrides

Endpoint	Free	Pro	Enterprise	Cost Multiplier
/mcp/execute	20/min	200/min	2,000/min	5x / 2x / 1x
/api/sessions	10/min	100/min	1,000/min	10x / 5x / 1x

3. Integration Guide

Quick Start

```
// 1. Import the rate limiter integration module
import {
  initializeRateLimiter,
  getRateLimiterMiddleware,
  getRateLimiterMetrics
} from './middleware/rateLimiterIntegration.js';

// 2. Initialize during application startup
await initializeRateLimiter();

// 3. Apply middleware to protected routes
app.use('/mcp/', getRateLimiterMiddleware());
app.use('/api/', getRateLimiterMiddleware());

// 4. Add metrics endpoint
app.get('/api/rate-limit/metrics', (req, res) => {
  res.json(getRateLimiterMetrics());
});

// 5. Graceful shutdown
process.on('SIGTERM', async () => {
  await closeRateLimiter();
  process.exit(0);
});
```

Environment Variables

Variable	Description	Default
REDIS_URL	Redis connection string	redis://localhost:6379
RATE_LIMIT_FAILOVER	Failover strategy	fail-closed

Response Headers

Header	Description	Example
RateLimit-Limit	Maximum requests per window	100
RateLimit-Remaining	Requests remaining in window	42
RateLimit-Reset	Unix timestamp when window resets	1702814460
Retry-After	Seconds until next request allowed	45
X-RateLimit-*	Legacy headers (backward compat)	Same as above

4. Security Audit Results

A comprehensive security audit was performed on the rate limiter implementation. The following areas were analyzed:

Category	Status	Details
API Key Handling	SECURE	SHA-256 hashing, no plaintext storage
Redis Connection	SECURE	Supports TLS via REDIS_URL, auth included
Input Validation	SECURE	All inputs sanitized, hashed before use
DoS Protection	SECURE	Cache size limits, LRU eviction
Information Disclosure	LOW RISK	Tier info in responses (acceptable)
Race Conditions	MITIGATED	Atomic Lua scripts in Redis
Memory Safety	FIXED	Added LRU eviction to tier cache

Vulnerabilities Checked

✓ SQL Injection - N/A (no SQL queries)
✓ XSS - N/A (JSON-only responses)
✓ CSRF - N/A (API key authentication)
✓ Timing Attacks - Mitigated by key hashing
✓ Denial of Service - Protected by rate limiting itself
✓ Redis Injection - Prevented by parameterized Lua scripts

5. API Reference

RedisRateLimiter Class

Method	Parameters	Returns	Description
initialize()	None	Promise<boolean>	Connect to Redis, load Lua script
middleware()	None	Express middleware	Create rate limiting middleware
getTier(apiKey)	string	Promise<string>	Get tier for API key
setTier(apiKey, tier)	string, string	Promise<boolean>	Set tier for API key
getMetrics()	None	object	Get current metrics
close()	None	Promise<void>	Close Redis connection

Error Response Format

```
{
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Rate limit exceeded. You have made too many requests.",
    "type": "https://api.example.com/errors/rate-limit-exceeded"
  },
  "rateLimit": {
    "limit": 100,
    "remaining": 0,
    "reset": 1702814460,
    "retryAfter": 45,
    "tier": "free"
  },
  "requestId": "req_1702814415_abc123",
  "timestamp": "2025-12-16T22:20:15.000Z",
  "help": {
    "message": "Please wait 45 seconds before making another request.",
    "documentationUrl": "https://docs.api.example.com/rate-limits"
  }
}
```

6. Metrics & Monitoring

The rate limiter exposes Prometheus-ready metrics for monitoring. Access metrics via the /api/rate-limit/metrics endpoint.

Metric	Type	Description
totalRequests	Counter	Total requests processed
allowedRequests	Counter	Requests allowed through
deniedRequests	Counter	Requests denied (429)
redisErrors	Counter	Redis connection errors
failoverActivations	Counter	Failover mode activations
requestsByTier	Counter	Requests per tier (free/pro/enterprise)
redisConnected	Gauge	Redis connection status
failoverCacheSize	Gauge	Current failover cache size
allowRate	Gauge	Percentage of allowed requests
denyRate	Gauge	Percentage of denied requests
requestsPerSecond	Gauge	Current request throughput

Recommended Alerts

High Deny Rate (> 10%): Indicates potential abuse or misconfigured limits
Redis Disconnected (redisConnected = false): Rate limiter in failover mode
High Failover Cache (> 5000 entries): Memory pressure during Redis outage
Error Rate Spike (> 5 errors/min): Redis connection issues