Index

Introduction

This document provides a detailed guide on implementing rate limiting for various tenant types within an **Node.js** application. Rate limiting is crucial for maintaining system stability, preventing misuse, and ensuring equitable resource distribution. It focuses on setting up dynamic rate limits tailored for basic, professional, and enterprise tenants. It includes configuring the Express.js server, organizing API routes, setting up middleware for rate limiting, and strategies for testing. By following these guidelines, This implementation enhances application security, optimize performance, and improve user satisfaction.

Installing and Configuring Redis on macOS

Step 1: Install Redis using Homebrew

Install Homebrew

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

Install Redis using Homebrew:

brew install redis

Step 2: Start and Verify Redis Server

1. **Start Redis Server**:
   - After installation, start the Redis server by running:

```
Redis-server
```

```
Last login: Fri Jul  5 12:25:37 on ttys000
sucharithareddyvem@Sucharithas-Air ~ % cd rate_limit
sucharithareddyvem@Sucharithas-Air rate_limit % redis-server
3369:C 05 Jul 2024 23:28:18.300 * oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
3369:C 05 Jul 2024 23:28:18.300 * Redis version=7.2.5, bits=64, commit=00000000, modified=0, pid=3369, just started
3369:C 05 Jul 2024 23:28:18.300 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
3369:M 05 Jul 2024 23:28:18.300 * Increased maximum number of open files to 10032 (it was originally set to 256).
3369:M 05 Jul 2024 23:28:18.300 * monotonic clock: POSIX clock_gettime
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 7.2.5 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 3369
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           https://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

3369:M 05 Jul 2024 23:28:18.302 # WARNING: The TCP backlog setting of 511 cannot be enforced because kern.ipc.somaxconn is set to the lower value of 128.
3369:M 05 Jul 2024 23:28:18.303 * Server initialized
3369:M 05 Jul 2024 23:28:18.303 * Ready to accept connections tcp
```

2. **Verify Redis Server Status**:

redis-cli ping

Step 3: Redis Configuration

Configuration File Location:

- By default, Redis on macOS installed via Homebrew uses a minimal configuration.
- Configuration file location: `/usr/local/etc/redis.conf`

Access Configuration File:

- nano /usr/local/etc/redis.conf

Restart Redis Server:

- brew services restart redis

Step 4: Interacting with Redis

1. **Using redis-cli**:

   redis-cli

File_Structure

```
project-root/
│
├── server.js                  # Main application file
├── routes/
│   └── api.js              # API routes file
├── middleware/
│   └── tenant_rate_limit.js # Middleware for rate limiting
└── tests/
    └── rate_limiter.test.js # Test cases for rate limiting middleware
```

• **Main Application Setup**: Explains the foundational setup of an Express.js server, crucial for integrating and deploying rate limiting functionalities.

• **API Routing**: Details the structuring of API endpoints and the integration of middleware to enforce rate limits per tenant type.

• **Middleware Configuration**: Discusses the implementation and customization of rate limiting middleware, emphasizing its role in enforcing request limits and managing client interactions.

• **Testing Strategy**: Outlines effective testing methodologies to validate the robustness and effectiveness of rate limiting measures across different tenant categories.

File Descriptions

Main Application File - `server.js`

Overview

The `server.js` file is the main entry point for the application. It sets up the Express server, integrates API routes, and starts the server on a specified port.

Contents

- **Imports**: Loads required modules (`express` and custom routes).
- **Express Application**: Creates an Express app instance.
- **Port Configuration**: Sets the port number for the server.
- **Route Integration**: Mounts API routes under the `/api` path.
- **Server Initialization**: Starts the server and listens on the configured port, logging a message upon successful start.

API Routes - `routes/api.js`

Overview

The `api.js` file defines the API routes for the application. It also integrates the rate limiting middleware to apply rate limits based on tenant types.

Contents

- **Imports**: Loads required modules (`express` and rate limiting middleware).
- **Router Setup**: Creates a router instance.
- **Middleware Application**: Applies the rate limiting middleware to all routes.
- **Route Definition**: Defines a GET route for the root path (`/`) that responds with a welcome message.

- **Export**: Exports the router for use in other parts of the application.

Rate Limiting Middleware - `middleware/tenant_rate_limit.js`

Overview

The `tenant_rate_limit.js` file implements the rate limiting logic based on tenant types. It defines different rate limits for basic, professional, and enterprise tenants and applies the appropriate limit based on the request header.

Contents

- **Imports**: Loads the `express-rate-limit` module.
- **Rate Limit Configurations**: Defines rate limit settings for different tenant types (basic, professional, enterprise).
- **Middleware Function**: Exports a middleware function that determines and applies the appropriate rate limit based on the `x-tenant-type` request header. Defaults to the basic rate limit if no tenant type is specified.

Tests for Rate Limiter - `tests/rate_limiter.test.js`

Overview

The `rate_limiter.test.js` file contains test cases to verify the behavior of the rate limiting middleware. It ensures that requests are correctly limited based on tenant types and that appropriate responses are returned when limits are exceeded.

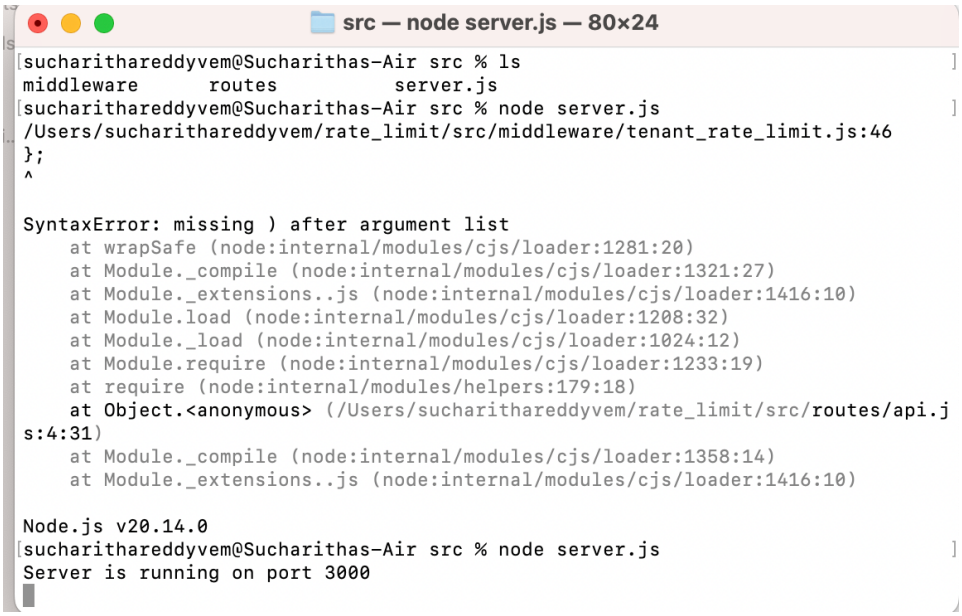
Contents

- **Imports**: Loads required modules (`supertest`, `express`, rate limiting middleware).
- **Setup**: Creates an Express application, applies the rate limiting middleware, and defines a root route for testing.
- **Jest Configuration**: Adjusts Jest's default timeout to accommodate delays in tests.
- **Test Suite Description**: Describes the overall test suite for the rate limiter middleware.
- **Before Each Test**: Clears all mocks to avoid shared state between tests.
- **Delay Function**: Defines a helper function to introduce delays in tests.
- **Tenant Test Cases**: Includes test cases for:
    - Enterprise Tenant: Verifies request limits and blocking behavior.
    - Professional Tenant: Verifies request limits and blocking behavior.
    - Basic Tenant: Verifies request limits and blocking behavior.

Working Overview :

1. `server.js`

- **Purpose**: Initializes the Express.js server, sets up middleware, and defines application-level configurations.
- **Key Responsibilities**:
  - Creates an instance of Express.js (`const app = express();`).
  - Configures middleware, such as body parsers, CORS handling, and logging.
  - Mounts API routes and other application-specific routes.
  - Starts the server listening on a specified port (`app.listen(PORT, () => { ... });`).



2. `routes/api.js`

Purpose: Defines the application's API endpoints and their corresponding logic.

- **Key Responsibilities**:
  - Imports necessary modules (`express`, controllers, middleware).
  - Defines route handlers for various HTTP methods (GET, POST, etc.).
  - Uses middleware functions for specific route protection or processing (`router.use(...)`).
  - Sends appropriate HTTP responses (`res.send()`, `res.json()`) based on business logic.

3. `middleware/tenant_rate_limit.js`

   Purpose: Implements rate limiting functionality to control the number of requests per tenant type.

- **Key Responsibilities**:
  - Intercepts incoming requests (`req`) before they reach route handlers.
  - Identifies the tenant type based on headers or parameters (`x-tenant-type`).
  - Applies rate limits (requests per minute/hour) based on the tenant's classification (basic, professional, enterprise).
  - Responds with HTTP status `429 Too Many Requests` and an error message when the rate limit is exceeded.

4. `tests/rate_limiter.test.js`

   Purpose: Validates the functionality of the rate limiting implementation through automated tests.

- **Key Responsibilities**:
  - Uses testing framework (e.g., Jest, Mocha) to define and execute test cases.
  - Sets up mock requests to simulate different tenant types and request scenarios.
  - Tests endpoints defined in `api.js` to ensure rate limits are enforced correctly.
  - Validates expected responses and behaviors:
    - **Allow Requests**: Ensures requests under the rate limit (e.g., 200 OK).
    - **Block Requests**: Verifies requests over the rate limit (e.g., 429 Too Many Requests).

```
Last login: Fri Jul  5 23:30:16 on ttys001
sucharithareddyvem@Sucharithas-Air ~ % cd rate_limit
sucharithareddyvem@Sucharithas-Air rate_limit % ls
Implement Per-Tenant Rate Limiting in Node.js Express API (1).docx
Index.docx
node_modules
package-lock.json
package.json
package.json.save
src
test
~$Index.docx
sucharithareddyvem@Sucharithas-Air rate_limit % npm test

> rate_limit@1.0.0 test
> jest

 PASS  test/rateLimiter.test.js (301.946 s)
  Rate Limiter Middleware
    Enterprise Tenant
      ✓ should allow requests under the rate limit for enterprise tenant (355 ms)
      ✓ should block requests over the rate limit for enterprise tenant (60308 ms)
    Professional Tenant
      ✓ should allow requests under the rate limit for professional tenant (60299 ms)
      ✓ should block requests over the rate limit for professional tenant (60229 ms)
    Basic Tenant
      ✓ should allow requests under the rate limit for basic tenant (60248 ms)
      ✓ should block requests over the rate limit for basic tenant (60207 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        301.973 s
Ran all test suites.
Jest did not exit one second after the test run has completed.

'This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jest with `--detectOpenHandles` to troubleshoot this issue.
```

Testing Workflow Example

1. **Setup**:
    - Initialize mock data, environment configurations, and server instances.
2. **Test Cases**:
    - **Enterprise Tenant**:
        - Verify requests under the rate limit (`for` loop with valid requests).
        - Validate blocking of requests over the rate limit (`await request(app).get(...)` exceeding the limit).
    - **Professional Tenant**:
        - Similar tests as above with different rate limits and request volumes.
    - **Basic Tenant**:
        - Ensure rate limits are properly applied for basic tenants.
3. **Assertions**:
    - Use assertions (`expect(...)`) to check HTTP status codes, response bodies, and error messages.
    - Clear mocks (`jest.clearAllMocks()`) between tests to maintain isolation.
4. **Execution**:
    - Run tests (`npm test` or specific test command) to execute all defined test cases.

Conclusion

By integrating these files and following this testing approach, which we can ensure that the rate limiting functionality operates effectively across different tenant types in the Node.js application. This process helps maintain application stability, optimize resource usage, and enhance overall user experience by preventing abuse and ensuring fair access to resources.