**Documentation : Node Assignment: User Task Queuing with Rate Limiting**

## Overview

This project sets up a scalable Node.js API with two worker processes to handle incoming tasks efficiently. It features rate limiting to control how frequently tasks can be submitted and uses a Redis-based queue to manage task processing. Task completions are logged to a file.

## Setup Instructions

1. **Install Dependencies**

npm install -y

npm i nodemon

npm i express-rate-limit

npm i express

npm i redis

npm i ioredis

1. **Set Up Redis**

Make sure Redis is installed and running . Follow the Redis installation guide if needed.

1. **Start the Application**

nodemon index.js

This will start the Node.js cluster with two workers.

## Usage Instructions

### API Endpoint

* **URL**: POST /api/v1/task in Postman
* **Request Body**:

json

Copy code

{

"user\_id": "123"

}

* **Response**:

json

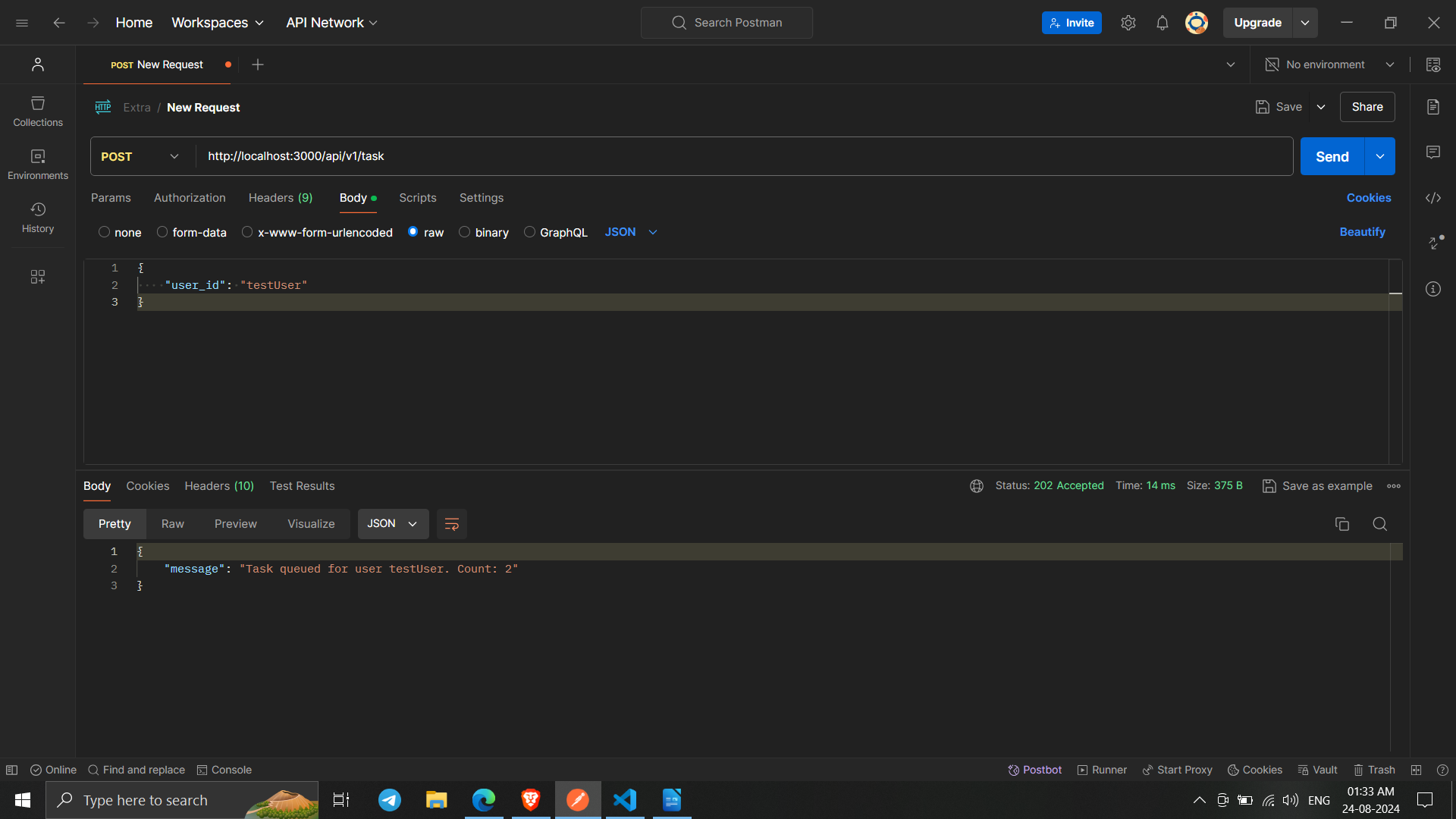
Copy code

{

"message": "Task queued for user 123. Count: <task-count>"

}

* **Details**: Submits a task for the user. Rate limits are enforced at 1 task per second and 20 tasks per minute per user. Tasks exceeding these limits are queued.



### After the 20 request :

### 

### Log File

Task completions are recorded in logs/task.log with entries like:

<user\_id> - task completed at - <timestamp>

## Approach

### ****1. Cluster****

* **Goal**: Improve performance and reliability by using multiple Node.js processes.
* **Implementation**: The application uses Node.js's built-in cluster module. This allows us to fork multiple worker processes (in this case, two) that can handle incoming requests simultaneously. The master process manages these workers and automatically replaces any that crash, ensuring the application remains available and responsive.

### 2. ****Rate Limiting****

* **Goal**: Prevent abuse by controlling how frequently tasks can be submitted per user.
* **Implementation**: Rate limiting is handled using the express-rate-limit library. We configured it to limit each user to:
  + **1 task per second**
  + **20 tasks per minute**

This is achieved by using a custom key generator based on the user\_id in the request body. The rate limiter middleware is applied to the /api/v1/task route to enforce these limits.

### 3. ****Task Queueing****

* **Goal**: Manage and process tasks efficiently, even when they exceed rate limits.
* **Implementation**: Tasks are queued using Redis, which is a high-performance, in-memory data structure store.
  + **Adding Tasks**: When a task is received, it’s added to a Redis list specific to the user.
  + **Retrieving Tasks**: A worker function retrieves tasks from Redis and processes them in the order they were received.

This approach helps in managing tasks asynchronously and ensures that even if tasks are rate-limited, they are processed in a timely manner.

### 4. ****Task Processing****

* **Goal**: Execute tasks and log the results.
* **Implementation**: Task processing is performed by worker processes. Each worker periodically checks Redis for tasks:
  + **Processing Tasks**: Each task is executed (in this case, simply logging a message with the user ID and timestamp).
  + **Logging**: Task completion is logged to a file to keep a record of when each task was processed. The logging mechanism ensures that the task log file is created if it doesn't already exist and appends each new entry.

## Assumptions

### 1. ****Redis Configuration****

* **Assumption**: Redis is assumed to be installed and running on the default local configuration (localhost).
* **Details**: The application uses Redis for managing task queues. By default, it expects Redis to be accessible at localhost on the default port (6379). If Redis is hosted on a different server or port, configuration changes are needed in the Redis client setup. Ensure Redis is properly installed and operational to avoid connection issues.

### 2. ****Node.js Environment****

* **Assumption**: Node.js is installed and properly configured on the system where the application is deployed.
* **Details**: The application relies on Node.js for running the server and processing tasks. It is assumed that a compatible version of Node.js is installed, as well as npm (Node Package Manager) for managing dependencies. Compatibility issues with Node.js versions could affect the functionality of the application, so it is crucial to use a supported version.

### 3. ****File System Access****

* **Assumption**: The application has the necessary permissions to create directories and write files for logging.
* **Details**: The application logs task completions to a file located at logs/task.log. It assumes that the process has write permissions to the file system to create and append to this log file. If the application runs in a restricted environment (such as a container with limited file system access), adjustments to permissions or file paths may be needed.

### 4. ****Rate Limits****

* **Assumption**: The rate limits (1 task per second and 20 tasks per minute) are strictly enforced per user.
* **Details**: The rate limiting mechanism is configured to ensure that each user can only submit tasks within these constraints. It assumes that the implementation correctly tracks and enforces these limits based on the user\_id provided in the request. If the rate limits are not enforced correctly, users might either exceed the limits or face unnecessary restrictions. The system should be tested to confirm that the rate limiting works as intended.

### 5. ****Task Processing Interval****

* **Assumption**: Tasks are processed every second, which is managed by the setInterval function in the task worker.
* **Details**: The application processes tasks at a fixed interval (every second). This approach assumes that task processing and rate limiting are well-balanced. If the processing interval or task load changes significantly, adjustments may be needed to ensure timely processing and compliance with rate limits.

### 6. ****Error Handling****

* **Assumption**: The application handles errors and edge cases effectively.
* **Details**: The application includes basic error handling for JSON parsing errors, Redis operations, and file system issues. It assumes that these mechanisms are sufficient to manage common issues. For more complex error scenarios or high-availability requirements, additional error handling or monitoring might be necessary.

### 7. ****Cluster Management****

* **Assumption**: Node.js clustering is used to improve performance and reliability.
* **Details**: The application uses Node.js's clustering capabilities to run multiple worker processes. It assumes that the cluster setup will provide the desired performance improvements and fault tolerance. Proper testing should be done to ensure that clustering meets the application's scalability and reliability needs.