Documentation for Node.js Application with Redis for Task Queuing and Rate Limiting

Problem Statement: User Task Queuing with Rate Limiting

Objective: Develop a Node.js API that manages user tasks with specific rate limits. The API should enforce a maximum of **1 task per second** and **20 tasks per minute** for each user. If a user exceeds these limits, their tasks should be queued and processed later according to the rate limits.

Key Requirements:

1. **Node.js API Cluster**: Set up a Node.js API server with clustering enabled to handle concurrent requests and distribute the load across multiple processes.
2. **Rate Limiting**: Implement a rate-limiting mechanism to ensure that each user can only perform a certain number of tasks per second and per minute.
3. **Task Queueing**: When a user exceeds the rate limit, queue their tasks and process them when allowed by the rate limit.
4. **Task Logging**: Log the completion of each task with the user ID and timestamp in a log file.
5. **Error Handling and Resilience**: Ensure the system can handle errors gracefully and is resilient to unexpected shutdowns.
6. **Use of Redis**: Utilise Redis to manage rate limits and task queues efficiently.

Overview of Application Components

1. Clustering and Server Setup (`server.js`)

- **Purpose**: The server setup file is responsible for initialising the Node.js application using clustering to handle multiple requests concurrently. It forks multiple worker processes to distribute incoming traffic.
- **Functionality**:
    - Sets up a master process to manage worker processes.
    - Spawns a specified number of worker processes (in this case, two replicas).
    - Handles incoming HTTP requests to a specific route (`/task`) and delegates processing to the task handler.

2. Task Handling and Rate Limiting (`taskHandler.js`)

- **Purpose**: This module manages task processing, rate limiting, and queuing.
- **Functionality**:
    - **Rate Limiting**: Uses Redis to implement rate limiting per user. It keeps track of task requests per user on a per-second and per-minute basis using Redis keys with expiration times.
    - **Task Queueing**: If the rate limit is exceeded, tasks are queued in Redis. Queued tasks are processed in the order they were received, respecting the rate limits.
    - **Logging**: Each task completion is logged to a file (`task_logs.txt`) with a user ID and timestamp.
    - **Graceful Shutdown**: Listens for application termination signals to close Redis connections cleanly, preventing data loss and ensuring smooth shutdown.

3. Log File (`task_logs.txt`)

- **Purpose**: This file records the log entries for each task processed by the API.
- **Content**: Each line contains information about a completed task, including the user ID and the timestamp when the task was completed.

How to Set Up and Run the Application

1. **Install Dependencies**:
    - Ensure that all required Node.js dependencies (`express`, `ioredis`) are installed.
    - Use command- npm install
2. **Start Redis Server**:
    - Run a Redis server instance on port 6380 or any other specified port as configured in your application.
    - Use command- npm start
3. **Run the Node.js Application**:
    - Start the application using Node.js. The clustering mechanism will fork the necessary worker processes to handle incoming requests.
4. **Test the API**:
    - Use a tool like Postman or `cURL` to send requests to the `/task` endpoint with a JSON payload containing a `user_id`.
    - Observe the behaviour when making multiple requests in quick succession to test rate limiting and queuing.
5. **Check Logs**:
    - Review the `task_logs.txt` file to see the logged task completions. Ensure that all tasks are processed according to the rate limit and that no requests are dropped.

Error Handling and Resilience

- The application is designed to handle errors gracefully. In case of a Redis connection failure or other exceptions, the application logs errors and ensures the API remains responsive.
- The clustering setup ensures that even if a worker process crashes, the master process will create a new worker, maintaining high availability.

Summary

This Node.js application effectively manages user tasks with strict rate limits and queues tasks as needed using Redis. It provides a robust, scalable solution for handling concurrent task processing while enforcing user-specific rate limits and ensuring tasks are processed in order.