Distributed Rate Limiter

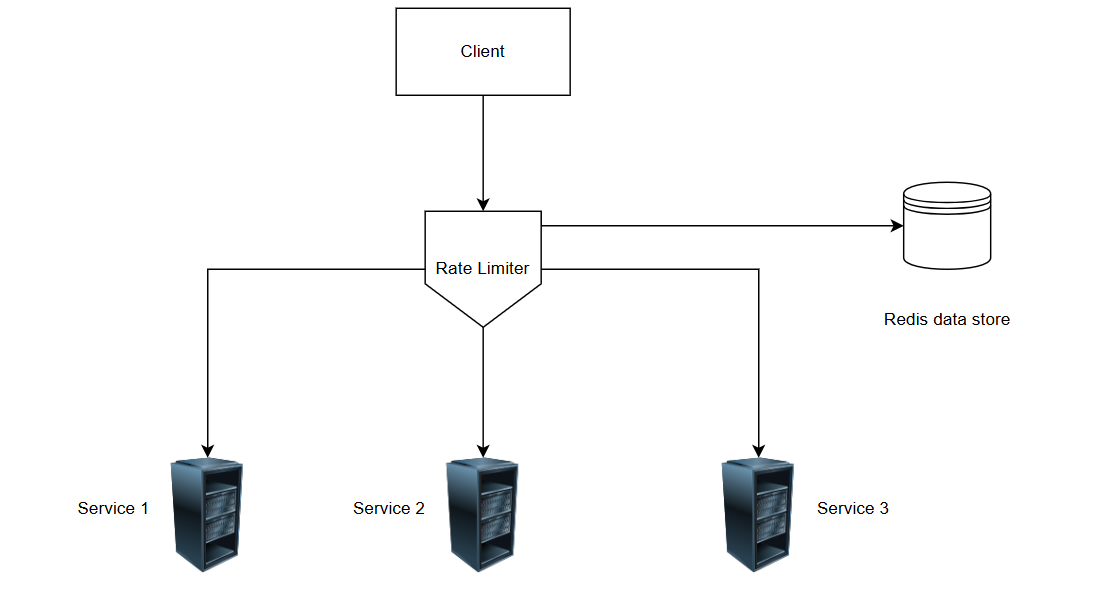
****

Figure: - 1 Distributed rate limiter diagram

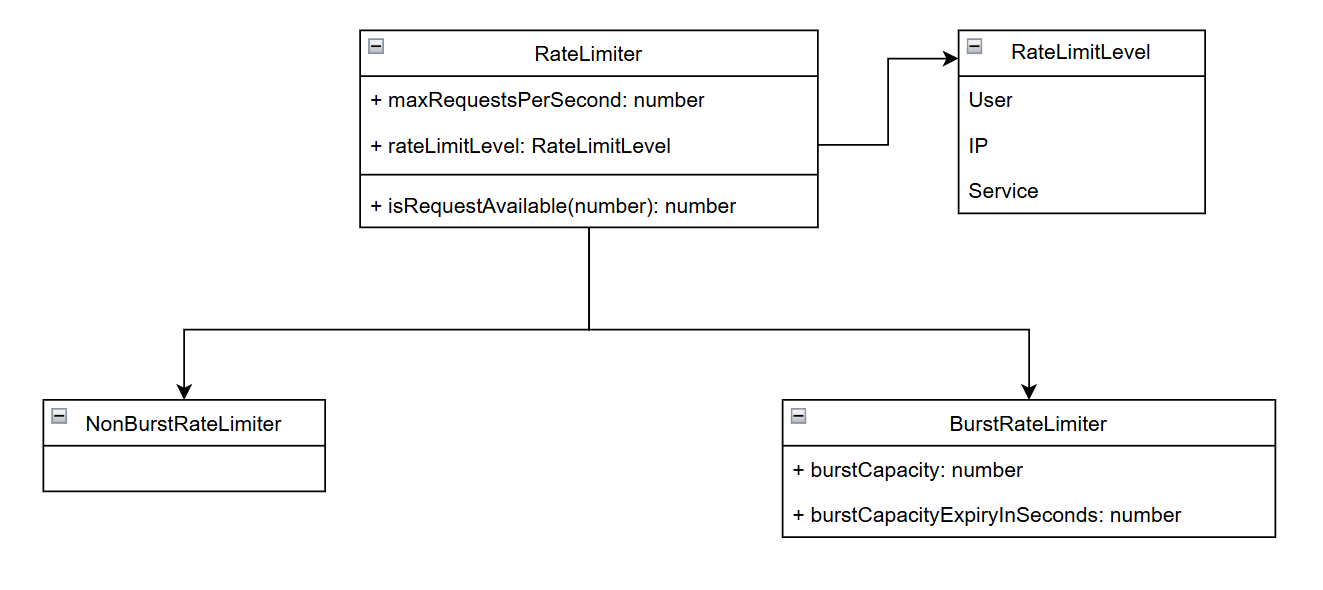
****

Figure: - 2 Class Diagram

* **Technology: -**
  + Implemented the backend using Node.js and developed a testing UI with React.js.
  + Utilized Redis as the data store.
* **How to use: -**
  + Integrate the rate limiter by injecting it as middleware with the desired rate limit configuration in each service.
  + The image below demonstrates how to apply the rate limiter to any service.



* **How to run project: -**
  + The application is containerized, allowing you to start it by running **docker-compose up.**
  + Alternatively, you can start the application by following these steps:
    - Run **npm install** in both the **root folder** and the **src/clientapp** directory.
    - Start the services with the following commands:
      * npm run start-service1
      * npm run start-service2
      * npm run start-service3
    - To start the testing UI, navigate to **src/clientapp** and run:
      * npm run dev
  + You can run all test cases using:
    - npm test
* **Implementation of services: -**
  + Implemented three services each with two endpoints:
    - api/service(service number)/nonBurst
    - api/service(service number)/burst
  + As the name suggest both endpoint implement non burst strategy & burst strategy respectively.
  + Example: -
    - api/service1/nonBurst
    - api/service1/burst
  + Three services hosted on different ports so can access each by following URLs:
    - locahost:3000
    - locahost:3001
    - locahost:3002
  + Three services also represent three rate limit levels, because each service implements one of the rate limit level like:
    - Service1 uses user level
    - Service2 uses IP level
    - Service3 uses service level
      * Service3 can be tested from the service one because to test service level, added two endpoints in service1.
        + /api/service1/nonBurst/callservice3
        + /api/service1/burst/callservice3
* **Implementation of UI: -**
  + Three Implemented simple UI to test different services & rate limiter.
  + Can access UI from **localhost:5173**
* **Endpoints: -**
  + http://localhost:3000/api/service1/nonBurst
  + http://localhost:3000/api/service1/burst
  + http://localhost:3000/api/service1/nonBurst/callservice3
  + http://localhost:3000/api/service1/burst/callservice3
  + http://localhost:3001/api/service2/nonBurst
  + http://localhost:3001/api/service2/burst
  + http://localhost:3002/api/service3/nonBurst
  + http://localhost:3002/api/service3/burst
  + http://localhost:5173
* **Implementation details: -**
  + Two rate-limiting strategies have been implemented: -
    - Max Requests per Second: Limits the number of requests to a set maximum per second.
    - Burst Capacity: Allows a temporary increase in the request limit to accommodate bursts.
  + First Strategy (Without Burst Capacity): Uses a sliding window algorithm:
    - Each time a request arrives, previous entries are removed based on their timestamps.
    - The number of requests within the last second is then counted.
    - Why Sliding Window is Needed:
      * Without it, tokens may expire simultaneously, causing Redis to reject expiration requests during concurrent operations.
      * Sliding window smooths this process, ensuring token expiration at different times.
  + Second Strategy (With Burst Capacity): Uses a "lazy refill" technique:
    - Rather than updating burst capacity every second, only the timestamp of the last request is stored.
    - Burst capacity is recalculated on the fly, reducing load on the data store and improving performance.
  + Extensibility:
    - The code is designed to support additional strategies. To add a new rate-limiting strategy, simply implement its service, and it can then be applied directly in the middleware.