

Distributed Rate Limiter

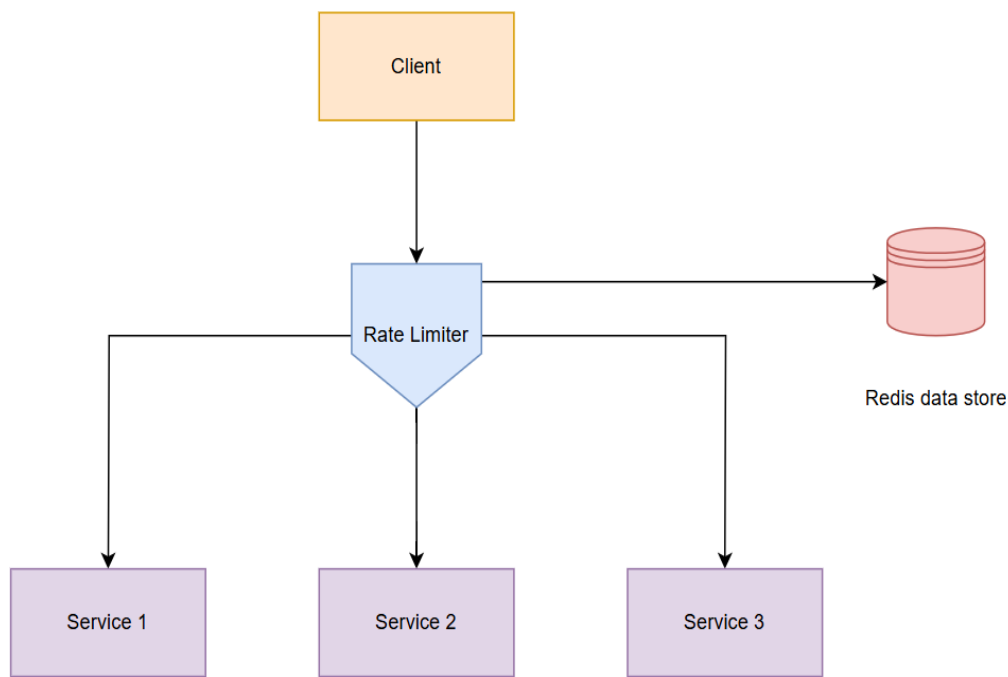


Figure: - 1 Distributed rate limiter diagram

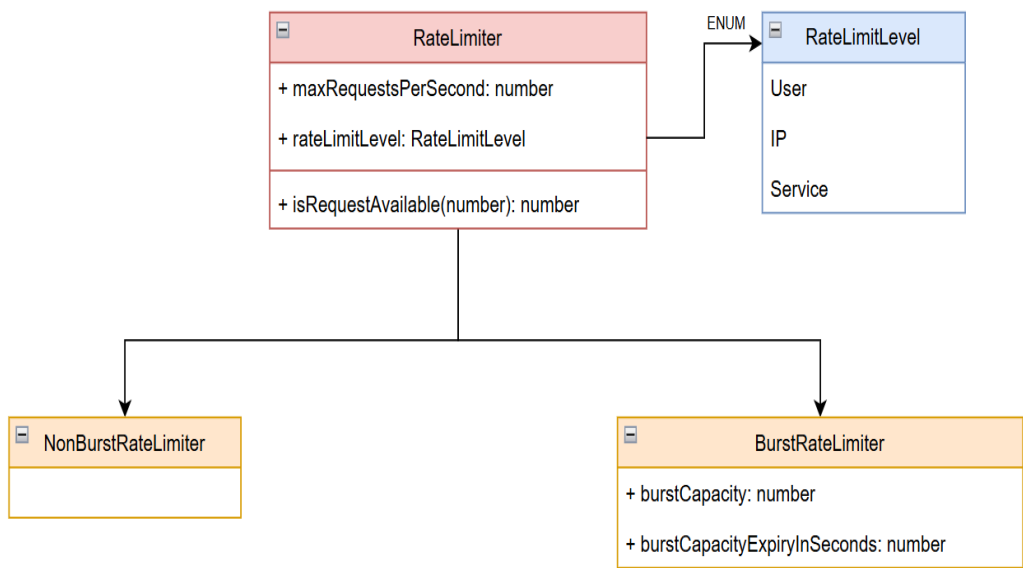


Figure: - 2 Class Diagram

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.

```
const rateLimiterConfigForNonBurst: RateLimiterConfig = new
RateLimiterConfig(10, RateLimitStrategy.NonBurstRateLimiter,
RateLimitLevel.User); //config

const rateLimiterConfigForBurst: RateLimiterConfig = new
RateLimiterConfig(10, RateLimitStrategy.BurstRateLimiter,
RateLimitLevel.User, 100, 60); //config

app.get("/api/service1/nonBurst", rateLimiterMiddleware
(rateLimiterConfigForNonBurst), (req, res) => {
  res.send({ message: "Accepted" });
})

app.get("/api/service1/burst", rateLimiterMiddleware
(rateLimiterConfigForBurst), (req, res) => {
  res.send({ message: "Accepted" });
})
```

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:

1. Run **npm install** in both the **root folder** and the **src/clientapp** directory.
2. Start the services with the following commands:
 - 1) **npm run start-service1**
 - 2) **npm run start-service2**
 - 3) **npm run start-service3**
3. To start the testing UI, navigate to **src/clientapp** and run: **npm run dev**
4. 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:
 - `localhost:3000`
 - `localhost:3001`
 - `localhost: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: -

- Implemented simple UI to test different services & rate limiter.
- Can access UI from `localhost:5173`

The screenshot displays the 'Rate Limiter Test UI' with three expandable sections. The first section, 'User Level Rate Limiter (Service 1)', is expanded and shows the following details:

- Strategy Selection:** Two radio buttons are present: 'Non burst strategy' (which is selected) and 'Burst Strategy'.
- Duration:** A text prompt asks 'Till how many seconds you want to make requests ?' followed by a value of '1 seconds' and a slider bar.
- Action:** A green button labeled 'Send Requests' is located below the duration settings.
- Results Table:** A table with four columns: 'Total Time', 'Total Request Sent', 'Accepted Requests', and 'Rejected Requests'. The data row shows values: 1.003, 119, 10, and 109 respectively.

The second section, 'IP Level Rate Limiter (Service 2)', and the third section, 'Service Level Rate Limiter (Service 3)', are currently collapsed.

| Total Time | Total Request Sent | Accepted Requests | Rejected Requests |
|------------|--------------------|-------------------|-------------------|
| 1.003 | 119 | 10 | 109 |

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.