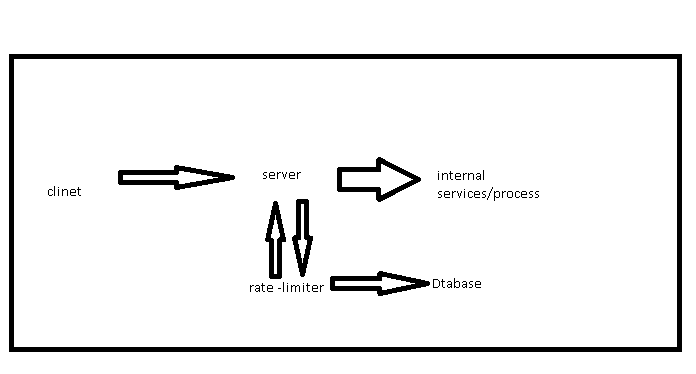
Rate limiting is a technique used in system design to control the rate at which incoming requests or actions are processed or served by a system. It imposes constraints on the frequency or volume of requests from clients to prevent overload, maintain stability, and ensure fair resource allocation.

**API Rate Limiting**: APIs often implement rate limiting to control the rate of requests from clients, ensuring fair access to resources and preventing abuse. For example, social media platforms limit the number of API requests per user per hour to prevent spamming and ensure system stability.

**High Level Design (HLD) to Design a Rate Limiter API** 

A rate limiter should generally be implemented on the server side rather than on the client side.

The overall basic structure of a rate limiter seems relatively simpler We just need a counter associated with each user to track how many requests are being same submitted in a particular timeframe. The request is rejected if the counter value hits the limit.

**--------------------------------------------------**

**Detailed System Design**

Architecture Overview

Client Requests: Clients send API requests which pass through the rate limiter before reaching the API server.

Rate Limiter: The rate limiter checks if the client has exceeded the request limit.

API Server: The API server processes the requests if they are within the allowed rate limit.

-------------------------------------------------

**Key Components**

API Gateway:

Acts as the entry point for all API requests. Integrates with the rate limiter to apply rate limiting rules.

Rate Limiter:

Can be implemented as a middleware in the API Gateway. Tracks requests and enforces limits.

Configuration Service:

Centralized service for managing rate limit configurations. Allows dynamic updates to rate limit settings without downtime.

Data Store:

Stores client request counts and rate limit configurations. Should support fast read and write operations, and handle high concurrency.

Notification Service:

Sends notifications or error messages when clients exceed the rate limit.

------------------------------------------------------------------------------------

Data Flow

Client sends a request to the API Gateway.

The API Gateway forwards the request to the Rate Limiter.

The Rate Limiter checks the request count for the client:

If within limit: Forwards the request to the API server.

If exceeded: Returns an error response to the client.

The API server processes valid requests and returns the response to the client.

Detailed Design

1. Rate Limiting Algorithm:

We choose” Leaky Bucket.”

”

2. Configuration Service

Use a key-value store to store rate limit settings.

Provide APIs to update configurations.

3. Data Store

Use Redis for its fast read/write capabilities and support for data expiration.

4. Notification Service

Use standard error handling mechanisms to send HTTP 429 responses.Include details about the limit and when the client can retry.

**Code VS system design mapping**

// version-0.1

/\* code sets up a basic rate limiter using Spring Boot and Redis. It includes a configurable limit, a mechanism to notify clients when the limit is exceeded, and a system designed to be highly available, low latency, and scalable\*/

Designing a rate limiter in Spring Boot to meet the specified functional and non-functional requirements involves integrating several components and leveraging Spring's powerful ecosystem.

Below is a detailed approach to designing a highly available, low-latency, and scalable rate limiter for an API.

### Functional Requirements

1. \*\*Limit the Number of Requests per Client within a Time Window\*\*:

- Use a rate limiting algorithm such as slide window (implementing through redis)

- Implement this logic using a combination of Spring Boot features and external tools like Redis for tracking requests.

2. \*\*Configurable Limit of Requests per Window\*\*:

- Create a configuration management system that allows setting and updating rate limits dynamically.

3. \*\*Notify Client When Threshold Is Exceeded\*\*:

- Return specific HTTP status codes and messages when the rate limit is exceeded.

### Non-Functional Requirements

1. \*\*Availability\*\*:

- Ensure the rate limiter is distributed and can failover between instances.

2. \*\*Low Latency\*\*:

- Optimize for minimal latency using in-memory data stores and efficient algorithms.

3. \*\*Scalability\*\*:

- Design for horizontal scaling to handle increasing loads.

### Design Steps

#### Step 1: Set Up Spring Boot Project

1. Initialize Project:

- Use Spring Initializr to create a Spring Boot project with dependencies like Web, Actuator, and Spring Data Redis (or other relevant data store).

2. Add Dependencies

- Add dependencies for Redis, Spring Cloud Config,

*Details code in File - pom.xml*

#### Step 2: Implement Rate Limiting Logic

1. \*\*Define Rate Limiter Configuration\*\*:

- Create a configuration class to hold rate limiting parameters.

*Details code in File -* RateLimiterConfig.java

2. \*\*Create Rate Limiter Service\*\*:

- Implement a service to handle rate limiting logic using Redis.

*Details code in File -* RateLimiterService.java

3. \*\*Rate Limiter Interceptor\*\*:

- Implement an interceptor to check rate limits before processing the request.

*Details code in File -* RateLimiterInterceptor.java

4. \*\*Register Interceptor\*\*:

- Add the interceptor to the Spring Boot application.

*Details code in File -* WebConfig.java

#### Step 3: Make Rate Limit Configurable

1. \*\*Spring Cloud Config\*\*:

- Use Spring Cloud Config to make rate limit settings configurable and dynamic.

*Details code in File -* application.yml

2. \*\*Reload Configuration\*\*:

- Implement a method to reload configuration without restarting the application.

*Details code in File -* ConfigController.java

#### Step 4: Ensure High Availability

-we Use a Redis cluster to distribute rate limiting data across multiple nodes.

#### Step 5: Optimize for Low Latency

we use Redis as an in-memory store for quick access to rate limiting data.

#### Step 6: Design for Scalability check

Horizontal Scaling- will deploy multiple instances of the Spring Boot application to handle increased load.

.