Rate Limiter Service Documentation

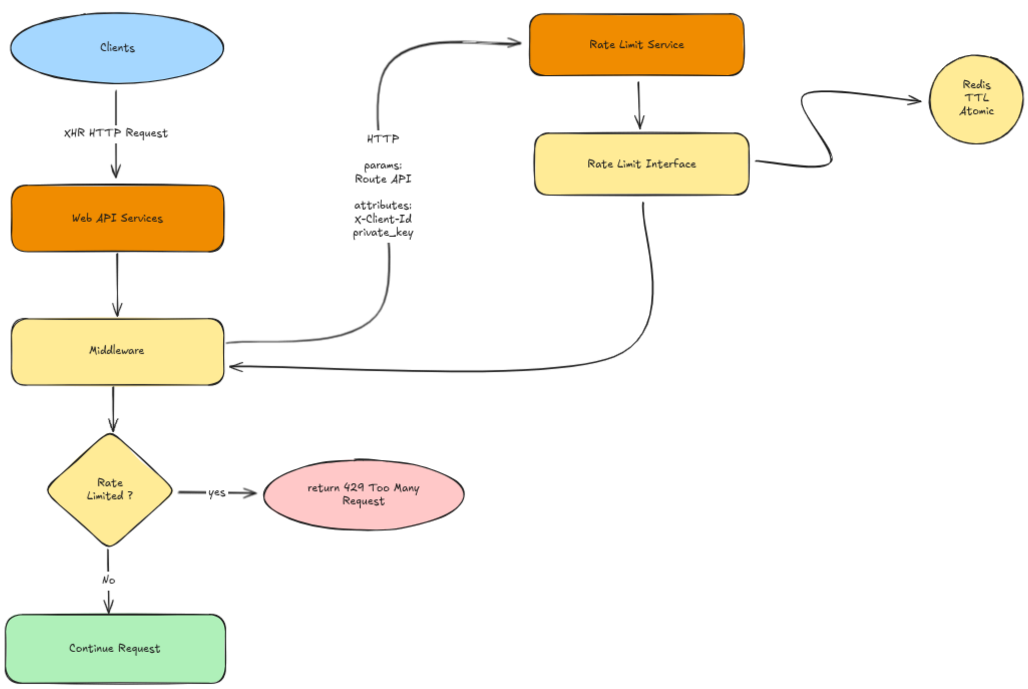
Write by Wildan Kurnia Candra

Regarding the detailed task description in this PDF, Doitpay Technical Test Documentation

This documentation covers the development and integration guide of a standalone Rate Limiter Service, designed to be used internally across web services to limit request flow on a per-client basis.

I've deployed the service too at `http://82.112.234.63:8080`

Github Repository https://github.com/user4xn/rate-limiter-service



It is built to control request traffic within a network architecture like a payment gateway, using the one of the bunch method “**fixed window**” algorithm as the task requirement.

This limiter is **not** meant to be exposed to public clients directly but rather to be used internally by services acting as middleware.

Key Features

- Fixed window rate limiting algorithm

- Support for multiple clients and routes
- Configurable limits per client and endpoint
- Thread-safe with optional distributed support (via Redis)
- Simple HTTP server interface

Use Case

Designed for environments like:

- **Payment gateway systems** where internal APIs must enforce quotas
- **Internal microservices** communicating with each other
- **Event-based batching systems** that control inflow to critical services

So, How It Works ?

🗺️ Architecture Flow

1. **Client** → Sends a request to the internal Web API service (e.g., initiate payment).
2. The **internal service** (e.g., Doitpay) uses middleware to **intercept the request**.
3. The middleware queries the **Rate Limiter Service** via HTTP.
4. The **Rate Limiter**:
   - Checks if the request exceeds the limit for the route and client.
   - Responds with `allowed: true` or `allowed: false`.
5. Based on the limiter response, the internal service either **processes or denies** the request.

   ⚠️ Note: Even when denied, the internal system may respond with a success status to keep the service-chain unbroken.
6. Redis is used for distributed key storage (client ID + route) with TTL for each fixed window.

🔐 Request Authentication

- All requests to the Rate Limiter must include a **valid API key** (used internally).
- Clients are uniquely identified via a `Client-ID` header.
- Each route is evaluated separately based on its client's configuration.

Let's Getting Started!

⚙️ Prerequisites

- Go installed

- Redis installed and running
- (Optional) Docker & Docker Compose installed

🧾 Clone the Repository

```
git clone https://github.com/user4xn/rate-limiter-service.git
cd rate-limiter-service
```

🧾 Configure the .ENV

```
cp .env.example .env
```

▶️ Run the Service

Option 1: Using Docker

```
docker-compose up -d --build
```

The service will be available at:

```
http://localhost:8080
```

> Make sure Redis is properly running in the container (or linked in docker-compose.yml).

Option 2: Run Manually

```
go run main.go
```

> This assumes Go and Redis are already installed on your machine.

🧪 Running Tests

Make sure Go is installed and you're in the root project directory.

```
go run main.go test
```

This runs:

- ✅ First-time access limit test

- 🔁 Concurrent burst simulation
- ⚙️ Configuration change test

📌 API Endpoints

🔍 1. Check Rate Limit

POST `/api/v1/rate/fixed-window`

```
curl --location --request POST 'http://{your_address}/api/v1/rate/fixed-window' \
--header 'Api-Key: {your_api_key}' \
--header 'X-Client-Id: {client_id}' \
--header 'Content-Type: application/json' \
--data-raw '{
    "route": "/api/v1/transactions"
}'
```

Example Response:

```
{
    "meta": {
        "message": "success",
        "code": 200,
        "status": "ok"
    },
    "data": {
        "status": "Allow", //OR Deny
        "limit": 100,
        "remain": 99,
        "reset_in_second": 41
    }
}
```

⚙️ 2. Set Client Configuration

PUT `/api/v1/rate/fixed-window/set`

```
curl --location --request PUT 'http://{your_address}/api/v1/rate/fixed-window/set' \
--header 'Api-Key: {your_api_key}' \
--header 'X-Client-Id: {client_id}' \
--header 'Content-Type: application/json' \
--data-raw '{
    "route": "/api/v1/transactions",
    "limit": 5,
```

```
  "window": 20
}'
```

Now How to Implement to Your Service ?

To integrate the rate limiter into your internal service, follow these steps:

1. 🧱 Setup Middleware in Your Internal Service

You need to create or update middleware logic that intercepts incoming HTTP requests and checks the rate limit before continuing.

Middleware Responsibilities:

- Extract `Client ID` and `route` from the request
- Call the Rate Limiter Service using `/api/v1/rate/fixed-window`
- Forward or reject the request based on the response

---

2. 📤 Call the Rate Limiter API

Use the `/api/v1/rate/fixed-window` endpoint to check if a request should be allowed.

Example:

```
curl --location --request POST 'http://localhost:8080/api/v1/rate/fixed-window' \
--header 'Api-Key: internal-service-secret' \
--header 'X-Client-Id: 123456' \
--header 'Content-Type: application/json' \
--data-raw '{
  "route": "/api/v1/transactions"
}'
```

- **Allowed Response:** `{ "allowed": true }`
- **Denied Response:** `{ "allowed": false, "reason": "rate limit exceeded" }`

You can map this into your service logic, allowing or blocking based on the result.

---

3. 🛠️ Optional: Set Custom Configuration

You can set client-specific limits using the `/api/v1/rate/fixed-window/set` endpoint.

Example:

```
curl --location --request PUT 'http://localhost:8080/api/v1/rate/fixed-window/set' \
--header 'Api-Key: internal-service-secret' \
```

```
--header 'X-Client-Id: 123456' \
--header 'Content-Type: application/json' \
--data-raw '{
    "route": "/api/v1/transactions",
    "limit": 10,
    "window": 60
}'
```

This is useful if different clients or routes require different rate-limiting strategies.

---

4. 🧪 Test the Integration

Simulate real requests from your internal service and verify that:

- Requests under the limit are allowed
- Requests exceeding the limit are blocked
- Configuration is stored and respected

Sample Go Middleware with Session-based `clientID`

```go
package middleware

import (
    "bytes"
    "encoding/json"
    "io"
    "net/http"
    "rate-limiter/pkg/util"

    "github.com/gin-gonic/gin"
)

type RateLimitRequest struct {
    Route string `json:"route"`
}

type RateLimitResponse struct {
    Meta struct {
        Message string `json:"message"`
        Code    int    `json:"code"`
        Status  string `json:"status"`
    } `json:"meta"`
    Data struct {
        Status      string `json:"status"` // "Allow" or "Deny"
        Limit       int    `json:"limit"`
```

```
        Remain        int    `json:"remain"`
        ResetInSecond int    `json:"reset_in_second"`
    } `json:"data"`
}

// RateLimiterMiddleware validates requests by contacting the Rate Limiter service.
// It assumes client ID is extracted from authenticated user/session context.
func RateLimiterMiddleware() gin.HandlerFunc {
    apiKey := util.GetEnv("API_KEY", "fallback")
    rateLimiterURL := util.GetEnv("RATE_LIMITER_URL", "http://localhost:8080")

    return func(c *gin.Context) {
        // Assume Auth middleware already sets user ID in context
        userID, exists := c.Get("userID") // change this key based on your auth system
        if !exists || userID == "" {
            c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{
                "status":  "failed",
                "code":    401,
                "message": "Unauthorized User",
            })
            return
        }

        // Prepare payload to Rate Limiter
        payload := RateLimitRequest{
            Route: c.FullPath(),
        }

        body, _ := json.Marshal(payload)
        req, err := http.NewRequest("POST", rateLimiterURL+"/api/v1/rate/fixed-window", bytes.NewB
uffer(body))
        if err != nil {
            c.AbortWithStatusJSON(http.StatusInternalServerError, gin.H{"message": "Error preparing r
ate limiter request"})
            return
        }

        req.Header.Set("Api-Key", apiKey)
        req.Header.Set("X-Client-Id", userID.(string))
        req.Header.Set("Content-Type", "application/json")

        resp, err := http.DefaultClient.Do(req)
        if err != nil || resp.StatusCode != http.StatusOK {
            c.AbortWithStatusJSON(http.StatusServiceUnavailable, gin.H{"message": "Rate limiter una
vailable"})
```

```
        return
    }
    defer resp.Body.Close()

    respBody, _ := io.ReadAll(resp.Body)
    var result RateLimitResponse
    if err := json.Unmarshal(respBody, &result); err != nil {
        c.AbortWithStatusJSON(http.StatusInternalServerError, gin.H{"message": "Invalid response
from rate limiter"})
        return
    }

    if result.Data.Status != "Allow" {
        c.AbortWithStatusJSON(http.StatusTooManyRequests, gin.H{
            "status":  "failed",
            "code":    429,
            "message": "Rate limit exceeded",
        })
        return
    }

    c.Next()
  }
}
```

Design Decisions I Made

This rate limiter was developed as part of a **technical interview assignment**. The design choices were inspired by how a payment gateway system, handles traffic control internally.

🔒 Internal-Only Service

Why:

To simulate a production-grade system like Doitpay, where services interact securely behind firewalls without exposing critical controls externally.

How:

- Services communicate with this rate limiter via **internal middleware**, not public endpoints.
- Every request to the limiter includes:
  - An `API-Key` (for internal authentication)
  - A `Client-ID` (unique to the calling service or user)
  - The specific API `route` being accessed

This ensures secure and traceable interactions across internal microservices.

🧠 Redis

Why:

Besides, it's a requirement, it supports **horizontal scaling** and consistent shared state across rate limiter instances.

How:

- Redis used for storing request counts with `INCR` and `EXPIRE`
- Ensures atomic operations across distributed nodes
- Optionally fallback to in-memory for single-instance setups

Redis enables the rate limiter to remain stateless and scalable.

⚙️ Safe Defaults: Fallback Config

Why:

To avoid any risk of unlimited traffic when a specific client config is missing.

How:

- Default rate limit (e.g., 100 requests/min) is applied if no config exists for a client+route pair
- Prevents accidental DDoS by new or misconfigured clients

This guarantees safe behavior across all endpoints, even during misconfiguration.

🔐 Internal Access Control via API Key

Why:

To ensure only authorized internal services can access the rate limiter.

How:

- Requires each request to include a valid `API-Key` header
- Key is validated before any rate-limiting logic is executed

This keeps the system secure and usage transparent across teams.

✅ Non-Intrusive Response: Always 200 OK

Why:

To prevent internal service failures due to rate limiting, while still enforcing limits.

How:

- The limiter **never returns an HTTP error** like `429`
- Instead, it responds with:

```
{
   "meta": {
      "message": "success",
      "code": 200,
      "status": "ok"
   },
   "data": {
      "status": "Deny",
      "limit": 100,
      "remain": 0,
      "reset_in_second": 41
   }
}
```

- The calling middleware decides how to respond to the original client

This design maintains smooth communication between internal systems and isolates concerns.

It Has Limitations, So Plan Future Improvements?

While the rate limiter is functional and production-ready for internal use, there are several **limitations** and potential **enhancements** that could be explored to improve flexibility, robustness, and scalability.

⚠️ Current Limitations

1. **Only Fixed Window Algorithm**
   - Currently supports only **fixed window** rate limiting.
   - Not suitable for bursty traffic or scenarios requiring smoother request distribution.
2. **No Persistent Storage**
   - All configurations and counters rely on Redis or in-memory storage.
   - Restarting without Redis will result in a loss of tracking data and configuration.
3. **No TLS/Encryption**
   - Communication is unencrypted over HTTP.
   - Should be deployed in a secure internal network or upgraded to HTTPS with proper certificates.
4. **Basic Auth Mechanism**
   - Uses a static API key validation approach.
   - Lacks more advanced service authentication like mTLS or OAuth.
5. **No Dashboard or Monitoring**

- No web UI or endpoint for visualizing traffic, rate limits, or blocked requests.
- Observability is limited without external integration.

6. **Latency Possibility**
    - Because each request involves an HTTP call to an external rate limiter, there's a small but noticeable latency penalty, especially under high traffic.

🚀 Future Improvements

1. **Sliding Window or Token Bucket Support**
    - Add additional algorithms for more precise rate control:
        - Sliding Window Log
        - Token Bucket (ideal for smoothing bursts)
2. **gRPC Support**
    - Add support for gRPC protocol to make integration easier for gRPC-based microservices.
3. **Fallback Modes**
    - Allow services to optionally receive 429 responses for easier enforcement on the middleware side.
4. **Improved Middleware SDK**
    - Publish Go (and potentially other language) SDKs for easier integration.
    - Abstract away API-key handling, retries, and rate-limit parsing.

Thankyou.