Coding Challenge for Backend Engineer Role (Golang Focus)

Challenge Title: *Design and Implement a Scalable Rate Limiting Service in Go*

Introduction

Hello! Thanks so much for taking the time to do our coding challenge. We hope that this won't take more than a few hours as we very much value your time.

We're looking for quality, conciseness, attention to detail, and following the requirements.

If you have any questions, feel free to reach out to Bennett here: badams@flourishsoftware.com

---

Problem Statement:

You are tasked with designing and implementing a rate limiter for an API that serves millions of requests daily. The rate limiter must ensure that:

1. Each client is limited to a maximum of **N requests per time window** (e.g., 100 requests per minute).
2. Excess requests within the time window are denied with an appropriate error message.
3. The system should be thread-safe, as multiple clients will make concurrent requests.
4. It should be flexible enough to configure different rate limits for different clients.

Additionally:

- The system must be memory-efficient and performant.
- Implementations that consider distributed systems (e.g., Redis-based rate limiting) are welcome but not mandatory for this exercise.

Deliverables:

1. A Go program implementing the rate limiter as a library or service.
2. Unit tests demonstrating the functionality of the rate limiter.
3. A short README explaining:
    - How the code works.
    - How to run the solution and tests.
    - Any assumptions or trade-offs you made during the implementation.
    - Approximately how much time was devoted to different parts of the exercise.

Please provide your solution as a link to a private GitHub repository. Or, if that is unavailable, a link to a `.zip` file.

Note: Please do not email a `.zip` file or other files as there is a good chance it will be caught by mail filters.

Requirements:

1. **Functionality:**
   - Implement a `RateLimiter` struct (or similar) with the following methods:

     ```go
     type RateLimiter interface {
         Allow(clientID string) bool
         SetRateLimit(clientID string, requests int, duration time.Duration)
     }
     ```

     - `Allow(clientID string) bool`: Returns `true` if the request is allowed, `false` otherwise.
     - `SetRateLimit(clientID string, requests int, duration time.Duration)`: Configures the rate limit for a specific client.
2. **Thread Safety:**
   - Ensure that the solution supports concurrent requests and handles thread safety appropriately.
3. **Performance:**
   - Optimize for high throughput and low latency.
4. **Extensibility:**
   - The implementation should allow for the possibility of plugging in a distributed backend (e.g., Redis) in the future.

Bonus Points:

- Implement a sliding window algorithm instead of a fixed time window.
- Include metrics or logging to show how many requests were allowed/denied.
- Provide insights on how you would scale this implementation in a distributed system (e.g., Redis, Kafka).

Sample usage to demonstrate how it works

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    // Create a new rate limiter
    rl := NewRateLimiter()

    // Set rate limits for different clients
    rl.SetRateLimit("client1", 5, time.Second)  // 5 requests per second
    rl.SetRateLimit("client2", 10, time.Second) // 10 requests per second
    rl.SetRateLimit("client3", 15, time.Second) // 15 requests per second

    // Simulate requests for multiple clients concurrently
    var wg sync.WaitGroup
    clients := []string{"client1", "client2", "client3"}

    for _, client := range clients {
        wg.Add(1)
        go func(clientID string) {
            defer wg.Done()

```

```go
            fmt.Printf("Starting requests for %s:\n", clientID)
            for i := 0; i < 20; i++ { // Simulate 20 requests per client
                if rl.Allow(clientID) {
                    fmt.Printf("[%s] Request %d: allowed\n", clientID, i+1)
                } else {
                    fmt.Printf("[%s] Request %d: denied\n", clientID, i+1)
                }
                time.Sleep(100 * time.Millisecond) // 10 requests per second pace
            }
        }(client)
    }

    wg.Wait()
    fmt.Println("All requests completed.")
}

```