Modelado y Programación — Proyecto 1

Diego Jardón, Pablo Trinidad.

1. Introducción

A continuación se presentan los detalles del proyecto 1 del curso de Modelado y Programación de la Facultad de Ciencias del semestre 2021-1 presentado por Diego Jardón y Pablo Trinidad.

El proyecto consta de un programa que a partir de un conjunto de aeropuertos de origen y destino, regrese los detalles del clima para cada uno de los pares de origen y destino de tal forma que esta información pueda ser presentada como información complementaria a un ticket de avión.

Se sugiere el uso de servicios web para obtener la información correspondiente del clima.

2. Descripción del problema

2.1. Descripción

Sea A el universo de aeropuertos disponibles (cada aeropuerto cuenta con latitud y longitud), y L un conjunto de pares ordenados $t_i = (a, b)$ tal que $a, b \in A$ y a representa el aeropuerto de origen de un vuelo y b el aeropuerto de destino, i.e: $L = \{t_1, t_2, ..., t_n\}$. Para cada $t_1 \in L$, se busca encontrar la información meteorológica relacionada a los aeropuertos de origen y destino de t_i. En particular temperatura, sensación térmica y humedad.

2.2. Naturaleza del problema

Sabemos por los requerimientos del problema que $0 \leq |L| \leq 3000$, por lo que $0 \leq |A| \leq 6000$. Sin embargo, se espera que estas ciudades estén repetidas entre los múltiples elementos de L, particularmente en el dataset brindado para realizar las pruebas (`dataset1.csv`) $|A| = X$.

2.3. Requerimientos no funcionales

1. Se espera que las comunicación con el servicio web utilizado para obtener la información del clima pueda ser realizada de manera concurrente.
2. El programa debe contar con un sistema básico de cache debe evitar realizar llamadas repetidas al servicio web y en su lugar utilizar las respuestas previamente obtenidas y almacenadas.

3. Análisis y búsqueda de herramientas

3.1. Proveedores de información meteorológica

3.1.1. Requerimientos

3.1.2. Opciones

3.1.3. Comparación

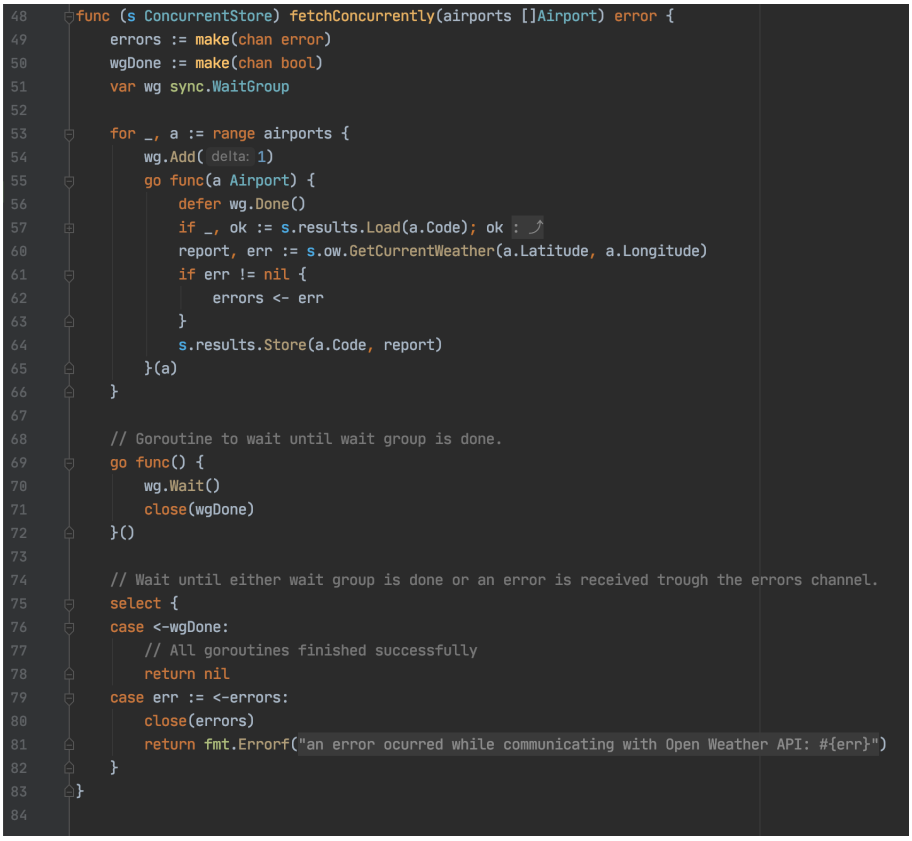
3.2. Sistema de cache

We used a fucking hash table mate, don't even trip.

3.3. Llamadas concurrentes

We used Go because ez p
Concurrency and cache model in a nutshell:

1. We make all HTTP requests run in parallel
2. Before making any HTTP request we first look up a hash table for previously fetched values (we use airport code as key), if key exists, then we don't, else, we do boiiiii.
3. FIN.



```go
func (s ConcurrentStore) fetchConcurrently(airports []Airport) error {
    errors := make(chan error)
    wgDone := make(chan bool)
    var wg sync.WaitGroup

    for _, a := range airports {
        wg.Add( delta: 1)
        go func(a Airport) {
            defer wg.Done()
            if _, ok := s.results.Load(a.Code); ok { …
            report, err := s.ow.GetCurrentWeather(a.Latitude, a.Longitude)
            if err != nil {
                errors <- err
            }
            s.results.Store(a.Code, report)
        }(a)
    }

    // Goroutine to wait until wait group is done.
    go func() {
        wg.Wait()
        close(wgDone)
    }()

    // Wait until either wait group is done or an error is received trough the errors channel.
    select {
    case <-wgDone:
        // All goroutines finished successfully
        return nil
    case err := <-errors:
        close(errors)
        return fmt.Errorf("an error ocurred while communicating with Open Weather API: #{err}")
    }
}
```

4. Solución

5. Futuras mejoras