Go Weather API Project - Flow and Code Overview

Project Summary

This Go project is a basic HTTP server that connects to the OpenWeatherMap API to fetch real-time weather data for a city provided via the URL.

The server exposes two endpoints:

1. /hello   - Returns a basic greeting message.
2. /weather/{city} - Fetches and returns weather data in JSON format for the specified city.

Code Flow Explanation

1. The `main()` function starts an HTTP server on port 8080 and sets up two routes: /hello and /weather/{city}.
2. For /weather/{city}:
   - It extracts the city name from the URL path.
   - Calls `query(city)` to get weather info from OpenWeatherMap API.
3. The `query()` function:
   - Loads the API key from `.apiConfig` file using `loadApiConfig()`.
   - Sends an HTTP GET request to OpenWeatherMap with the API key and city.
   - Parses the JSON response into a Go struct.
4. The response is encoded into JSON and sent back to the client.

Error handling is done using `http.Error` for client responses and Go's standard error return values.

Usage Instructions

How to Use:

1. Create a `.apiConfig` file in the same directory with content like:

```
   {
    "OpenWeatherMapApiKey": "your_api_key_here"
  }
```

2. Run the server:

```
go run main.go
```

3. Open a browser or use curl to test:

   - http://localhost:8080/hello
   - http://localhost:8080/weather/delhi

The response will be JSON containing the weather data for the specified city.

Source Code

```
package main

import (
 "encoding/json"
 "io/ioutil"
 "net/http"
 "strings"
)

//accesing the OpenWeatherMap API Key
type apiConfigData struct {
 OpenWeatherMapApiKey string `json: "OpenWeatherMapApiKey"`
}

type weatherData struct {
 Name string `json:"name"`
 Main struct {
  Kelvin float64 `json:"temp"`
 } `json: "main"`
}

//function to load api configuration from a file
func loadApiConfig(filename string) (apiConfigData, error) {
```

Go Weather API Project - Flow and Code Overview

```go
 bytes, err := ioutil.ReadFile(filename)

 // Check if there was an error reading the file
 if err != nil {
  return apiConfigData{}, err
 }

 var c apiConfigData
 //unmarshal means to convert the JSON bytes to a struct
 err = json.Unmarshal(bytes, &c) // Convert the JSON bytes to a struct
 if err != nil {
        return apiConfigData{}, err
    }
 return c, nil
}

func hello(w http.ResponseWriter,r *http.Request){
 w.Write([]byte("Hello from Go!\n"))
}

func query(city string) (weatherData, error) {
 apiConfig, err := loadApiConfig(".apiConfig")
 if err != nil{
  return weatherData{}, err
 }

 resp,       err       :=       http.Get("https://api.openweathermap.org/data/2.5/weather?APPID="+
apiConfig.OpenWeatherMapApiKey + "&q=" + city)
 if err != nil {
  return weatherData{}, err
 }

 defer resp.Body.Close()

 var d weatherData

 if err := json.NewDecoder(resp.Body).Decode(&d); err!=nil {
  return weatherData{}, err
 }
 return d, nil
}
```

Go Weather API Project - Flow and Code Overview

```
func main() {
 http.HandleFunc("/hello", hello)

 http.HandleFunc("/weather/",
 func(w http.ResponseWriter, r *http.Request){

  city := strings.SplitN(r.URL.Path, "/",3)[2]
  //so basically this like divides the path into 3 parts on the basis of "/" and gets the third element
  data, err := query(city)
  if err!=nil {
   http.Error(w, err.Error(), http.StatusInternalServerError)
   return
  }
  w.Header().Set("Content-Type","application/json; charset=utf-8")
  json.NewEncoder(w).Encode(data)
 })

 //create a new server and listen on port 8080
 http.ListenAndServe(":8080", nil)
 //Use:
 //http.Error --> // to send an error response to the client
 //log.Printf --> // to log info or errors on the server console
 //fmt.Printf --> // to print info or errors to the console


}
```