ECGR 4090/5090 Cloud Native Application Architecture
Lab 6: Accessing external service through public REST API

In this lab, we will programmatically access external services through public REST APIs. Cloud native application development relies on REST APIs to share data with external users. APIs provide a well defined interface for external users to access a service while hiding internal implementation details. Additionally, APIs allow incorporation of security principles such as authentication and authorization. As an example, Google's Places API is a service that returns information about places using HTTP requests. Places are defined within this API as establishments, geographic locations, or prominent points of interest.
Check out the details of the Google Places API here -
https://developers.google.com/maps/documentation/places/web-service/overview

RapidAPI hub provides an extensive collection of publicly available APIs for a plethora of services.
https://rapidapi.com/hub

REST APIs can be accessed programmatically (for example in Go) by making appropriate HTTP requests. The Google Places API is a paid service with a limited free tier. In this lab, we will access weather data through the free OpenWeather API. Typically, to access an API you will need to signup for the service, and obtain an API access key (a hex string). All API requests are made with the API key included for authentication. OpenWeather API returns data in JSON and XML format. We will parse the JSON data with Go, and display the relevant information to the user.

Head over to OpenWeather and signup (free).
https://openweathermap.org/api

You will be provided a default API key after verifying your email. The key is accessible through your account page.

Let's check what the APIs look like. You can see different types of services provided on the Weather API page including Current Weather Data, Hourly Forecasts for 4 days, Solar Radiation API etc. We will use the free Current Weather Data in this lab. Click on the Current Weather Data - API doc. You will see details on how to make an API call for this service. For example, to access weather data at a particular latitude and longitude,

https://api.openweathermap.org/data/3.0/onecall?lat={lat}&lon={lon}&exclude={part}&appid={API key}

The fields in curly braces are user inputs. The 3.0 denotes the API version.

An example response would be weather data in JSON format. Recollect that JSON is a human readable data format, where data is expressed as maps (key-values), and lists. Keys are strings.

```json
{
  "coord": {
    "lon": -122.08,
    "lat": 37.39
  },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 282.55,
    "feels_like": 281.86,
    "temp_min": 280.37,
    "temp_max": 284.26,
    "pressure": 1023,
    "humidity": 100
  },
  "visibility": 16093,
  "wind": {
    "speed": 1.5,
    "deg": 350
  },
  "clouds": {
    "all": 1
  },
  "dt": 1560350645,
  "sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0139,
    "country": "US",
    "sunrise": 1560343627,
    "sunset": 1560396563
  },
  "timezone": -25200,
  "id": 420006353,
  "name": "Mountain View",
  "cod": 200
}
```

Explanations of the different fields are provided on the API doc.
If you scroll further down, you will see data responses in the XML format. See how it compares to JSON.

OpenWeather also provides Geocoding APIs that allow you to access weather data for a particular city. The API looks like this -
https://api.openweathermap.org/data/2.5/weather?q=London&appid={API_KEY}

Try this in your browser. Insert the API_KEY from your *openweathermap.org* account
In this lab we will be using the Geocoding APIs to access weather information for an input city.

Go Command Line Interface (CLI)

Let's now write a Go command line tool that accesses the Open Weather API, parses the JSON data, and displays a brief weather information to the user. We will also see how to do Test Driven Development (TDD) for this project.

First the data types -
Client - (api key, url, and HTTP client from the net/http package)

```
type Client struct {
        APIKey     string
        BaseURL    string
        HTTPClient *http.Client
}
```

Client constructor (Recollect in Go constructor is not part of the class)-

```
func NewClient(key string) *Client {
        return &Client{
                APIKey:  key,
                BaseURL: "https://api.openweathermap.org",
                HTTPClient: &http.Client{
                        Timeout: 10 * time.Second,
                },
        }
}
```

We'll add two methods to the Client - (1) *formatURL()* that takes in the city as input, and generates the appropriate URL; and (2) *GetWeather()* that makes the GET request to the service with the URL, and calls a *ParseReponse()* helper method to parse the JSON data.

In this example, we are only interested in presenting 2 pieces of weather data to the user - the summary, and the current temperature.
We create a *Temperature* data type with a *Fahrenheit()* method that converts the Kelvin temperature returned by the API to Fahrenheit temperature.

```
type Temperature float64

func (t Temperature) Fahrenheit() float64 {
	return (float64(t) - 273.15)*(9.0/5.0) + 32.0
}

type Conditions struct {
	Summary     string
	Temperature Temperature
}
```

If you check out the example JSON response listed above, all we need is the main field from “weather” key which gives us the summary, and the temperature field from “main” key that gives us the temperature. We create a data structure to represent these response fields.

```
type OWMResponse struct {
	Weather []struct {
		Main string
	}
	Main struct {
		Temp Temperature
	}
}
```

The *ParseReponse()* method calls the encoding/json package to Unmarshal the JSON data into the *OWMResposne* data type, and then generates and returns the *Conditions* object.

A *RunCLI()* method is included for the user to run the client on the command line. The method parses command line arguments of user input using *os.Args*, and extracts the API key stored in an environment variable.
Note: You should never hard code API keys in your code. The code is typically checked in a code repository such as Github, and any user that accesses the code has access to your private key. Best practice is to store the key as an environment variable in the shell. On bash shell,

```
$ export OPENWEATHERMAP_API_KEY=”Your API key”
```
This is read from Go using the *os.Getenv()* method.

Test driven development

To test the Go WeatherAPI client, a mock response data is created in a *testdata* directory. Valid and invalid data are generated, to run Go tests. In weather_test.go, different test scenarios are included that use this mock data. These included TestParseResponse, *TestParseResposneEmpty, TestParseResposneInvalid, TestFormatURL, TestFormatURLSpaces, TestSimpleHTTP, TestGetWeather, and TestFahrenheit.* Note the extensive tests that need to be written to test out different code paths.

The complete code is available on Canvas. Please download the zip file. Test the code with go test. Then run the CLI from *main.go* and make sure the Go client works. Modify the include paths appropriate to your system. As a test, check the weather in Charlotte!

Acknowledgements: The code is adapted from the following repo - https://github.com/mr-joshcrane/weather_api

To do -
Extend the CLI to print out pressure, humidity and wind speed information as well. Extend the weather_test.go, *TestGetWeather()* method to add tests for these.