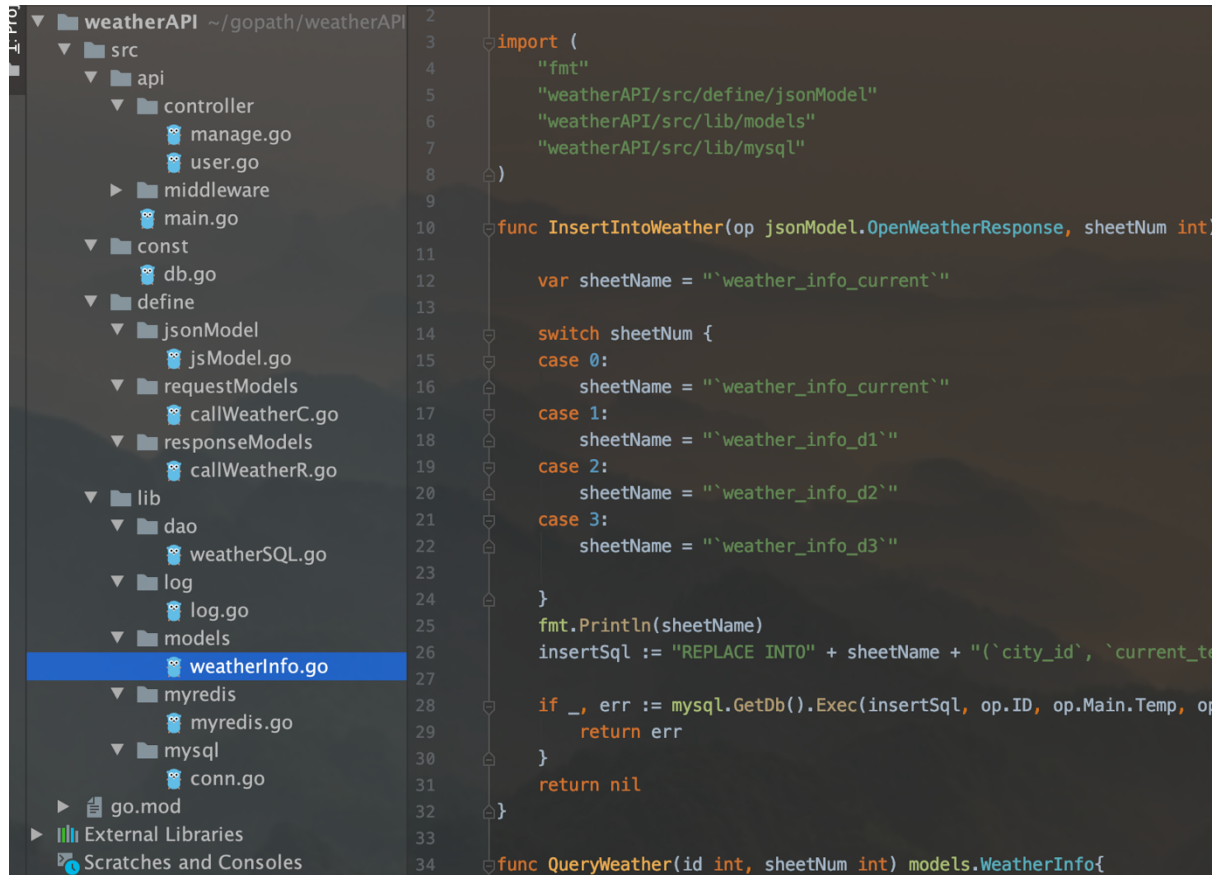


Weather API Doc

Codes Structure:



The image shows a code editor with a project structure on the left and Go code on the right. The project structure is as follows:

- weatherAPI ~/gopath/weatherAPI
 - src
 - api
 - controller
 - manage.go
 - user.go
 - middleware
 - main.go
 - const
 - db.go
 - define
 - jsonModel
 - jsModel.go
 - requestModels
 - callWeatherC.go
 - responseModels
 - callWeatherR.go
 - lib
 - dao
 - weatherSQL.go
 - log
 - log.go
 - models
 - weatherInfo.go
 - myredis
 - myredis.go
 - mysql
 - conn.go
 - go.mod
 - External Libraries
 - Scratches and Consoles

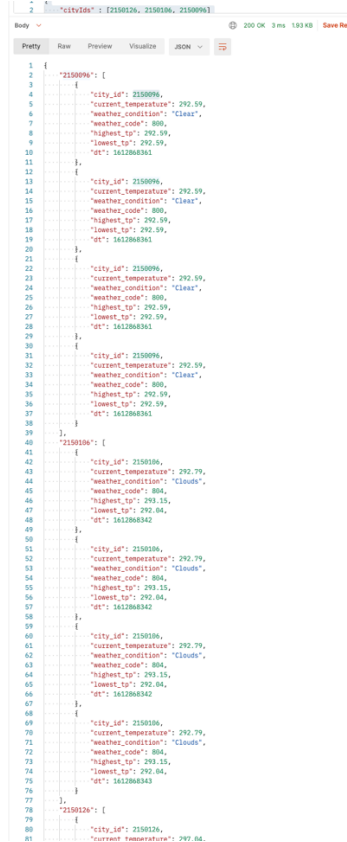
The Go code on the right is as follows:

```
2
3 import (
4     "fmt"
5     "weatherAPI/src/define/jsonModel"
6     "weatherAPI/src/lib/models"
7     "weatherAPI/src/lib/mysql"
8 )
9
10 func InsertIntoWeather(op jsonModel.OpenWeatherResponse, sheetNum int) {
11     var sheetName = "`weather_info_current`"
12
13     switch sheetNum {
14     case 0:
15         sheetName = "`weather_info_current`"
16     case 1:
17         sheetName = "`weather_info_d1`"
18     case 2:
19         sheetName = "`weather_info_d2`"
20     case 3:
21         sheetName = "`weather_info_d3`"
22     }
23
24     fmt.Println(sheetName)
25     insertSql := "REPLACE INTO" + sheetName + "(`city_id`, `current_t"
26
27     if _, err := mysql.GetDb().Exec(insertSql, op.ID, op.Main.Temp, op
28         return err
29     }
30     return nil
31 }
32
33 func QueryWeather(id int, sheetNum int) models.WeatherInfo{
```

APIs:

- Query weather

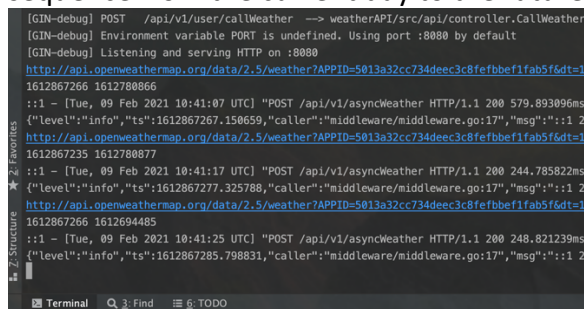
User requests a post with API to get a JSON package like:



the frontEnd can decode this JSON package as a list, which is a convenience for searching and save to list, and also swipe the screen to left or right

- Async weather via open weather API into MySQL

Just database design fulfills the free plan of OpenWeather. I can not get history data via API with multiple requests. The API gave the same weather, although I input a different timestamp. So the request weather API returns the same data but with a sequence from the current day to the future three days.



Database Implement:

I first use XORM to create tables with structs

```
type WeatherInfoCurrent struct {
    CityId          int    `xorm:"pk" json:"city_id"`
    CurrentTemperature float64 `xorm:"notnull" json:"current_temperature"`
    WeatherCondition string  `xorm:"notnull" json:"weather_condition"`
    WeatherCode     int    `xorm:"notnull" json:"weather_code"`
    HighestTP       float64 `xorm:"notnull" json:"highest_tp"`
    LowestTP        float64 `xorm:"notnull" json:"lowest_tp"`
    DT              int    `xorm:"updated" json:"dt"`
}

type WeatherInfoD1 struct {
    CityId          int    `xorm:"pk" json:"city_id"`
    CurrentTemperature float64 `xorm:"notnull" json:"current_temperature"`
    WeatherCondition string  `xorm:"notnull" json:"weather_condition"`
    WeatherCode     int    `xorm:"notnull" json:"weather_code"`
    HighestTP       float64 `xorm:"notnull" json:"highest_tp"`
    LowestTP        float64 `xorm:"notnull" json:"lowest_tp"`
    DT              int    `xorm:"notnull" json:"dt"`
}

type WeatherInfoD2 struct {
    CityId          int    `xorm:"pk" json:"city_id"`
    CurrentTemperature float64 `xorm:"notnull" json:"current_temperature"`
    WeatherCondition string  `xorm:"notnull" json:"weather_condition"`
    WeatherCode     int    `xorm:"notnull" json:"weather_code"`
    HighestTP       float64 `xorm:"notnull" json:"highest_tp"`
    LowestTP        float64 `xorm:"notnull" json:"lowest_tp"`
    DT              int    `xorm:"notnull" json:"dt"`
}

type WeatherInfoD3 struct {
    CityId          int    `xorm:"pk" json:"city_id"`
    CurrentTemperature float64 `xorm:"notnull" json:"current_temperature"`
    WeatherCondition string  `xorm:"notnull" json:"weather_condition"`
    WeatherCode     int    `xorm:"notnull" json:"weather_code"`
    HighestTP       float64 `xorm:"notnull" json:"highest_tp"`
    LowestTP        float64 `xorm:"notnull" json:"lowest_tp"`
    DT              int    `xorm:"notnull" json:"dt"`
}
```

I create four tables: weatherInfoCurtent, weatherInfoD1, weatherInfoD2, weatherInfoD3.

But in the implementation of data query and insert, I use native sql, because ORM structure is complicated, and I needed to write four structure for four tables each API(async and query)

Optional task:

I used Redis into Gin, but the result does not well

```
(Decex) x rongbaizhang@192-168-1-108 ~/PycharmProjects/SQLtask/weather python apiPerformance.py
0:00:05.750877 with redis 1000times
0:00:04.873221 without redis 1000times
(Decex) rongbaizhang@192-168-1-108 ~/PycharmProjects/SQLtask/weather python apiPerformance.py
0:01:03.406530 with redis 10000times
0:00:57.599475 without redis 10000times
(Decex) rongbaizhang@192-168-1-108 ~/PycharmProjects/SQLtask/weather python apiPerformance.py
0:11:26.807948 with redis 100000times
0:09:57.382788 without redis 100000times
(Decex) rongbaizhang@192-168-1-108 ~/PycharmProjects/SQLtask/weather python apiPerformance.py
```

```
} else {
    DataWeather := make(map[int][]models.WeatherInfo)
    conn := myredis.RedisDefaultPool.Get()
    defer conn.Close()
    for _, id := range stat.CityIds {
        s := make([] models.WeatherInfo, 0)
        redisKey := "WC" + strconv.Itoa(id)
        ret, err := redis.Bytes(conn.Do( commandName: "get", redisKey))
        weatherObj := []models.WeatherInfo{}
        if err != nil {
            for i := 0; i < 4; i++ {
                res := dao.QueryWeather(id, i)
                s = append(s, res)
            }
            retDate, _ := ffjson.Marshal(s)
            conn.Do( commandName: "setex", redisKey, 2000, retDate)

            DataWeather[id] = s
        } else {
            ffjson.Unmarshal(ret, &weatherObj)

            DataWeather[id] = weatherObj
        }
    }
}
```