MSCS 621L - Fall 2025 - R. O. Topaloglu

Weather Monitoring Framework
Project 2

Tyler DeLorey
tyler.delorey1@marist.edu

November 2nd, 2025

Contents

1 How to Setup Project

1.1 Installing Docker

This project uses Docker in order to set up everything correctly. I would recommend installing Docker Desktop (especially if you are on Windows), as that is what I used and it worked out. To learn how to install Docker Desktop, click the link here.

Now you should have installed Docker Desktop and created an account. You can type `docker version` in your CLI of choice to see if it installed correctly. If it did, in the directory of this project, if you aren't logged in, type `docker login` to login to your account.

1.2 Commands to Build Images and Run Containers

First of all, in your directory, make sure there exists the `proj2_image.tar` file, as that is the Docker Image.

In order to start, we must run the containers for Kafka, Prometheus, and Grafana using pre-built images by these different sources. All of these configurations were created in the `docker-compose.yml`. To run these containers in the background (detached), run the following command:

- `docker-compose up -d kafka -d prometheus -d grafana`

After the containers are set up and running, we can load the Docker Image. To load this Docker Image into your own Docker, use the command:

- `docker load -i proj1_image.tar`

However, because your API key is set in the `docker-compose.yml` file, loading the image won't be enough for the program to work. You must **set your OpenWeather API key in the `docker-compose.yml` file** under the `proj2` service. There, you can also change the file used, the number of workers in the worker pool, the alert thresholds, and more. Now you can rebuild the image using the command:

- `docker-compose build proj2`

Now you can run this program using the command:

- `docker-compose run --rm -it --service-ports proj2`
  - `--rm` makes sure the container is removed after we run it, since for this project and my implementation, the container doesn't need to run after the program ends.

  - `-it` allows user input to be read. This is useful because the program must stay running in order for the Prometheus metrics to be read and for Grafana to update properly. So, the user can press enter to end the program once they feel the need to.
  - `--service-ports` ensures that the ports used in this program (specified in the `docker-compose.yml` file) are actually up and running.

The above command will automatically use the FILE found in the `docker-compose.yml` file. If you want to change this file you can either:

- Rebuild the proj2 image (`docker-compose build proj2`) and run the program using the previous command
- Recommended: Use the command `docker-compose run --rm -it --service-ports -e FILE='file.txt' proj2`, replacing `file.txt` with the name of this new file. Make sure this .txt file was in the directory when proj2 was built!

If you want to shut everything down, you can run the following commands:

- `docker-compose down`, which shuts down and removes all containers
- `docker rmi apache/kafka prom/prometheus grafana/grafana proj2-proj2`, which removes all images
- `docker volume prune -a`, but **BE CAREFUL!** This will remove every volume currently in your Docker, which you might not want to do! The problem is that Kafka creates some volumes with hashed names, meaning it is difficult to select these volumes to remove. If you want to remove a volume by name, use the command: `docker volume remove vol_name`, replacing `vol_name` with the volume name.

If you want to save the image on your computer, run the command:

- `docker save -o proj2_image.tar proj2-proj2`

The next section has all of the needed information for the software architecture used in the code and why these decisions were made.

2 Explanations

2.1 Dockerfile

Below is my implementation of the Dockerfile, the file that is used (with the source code) to build the Docker Image for this program.

```
# -- BUILD STAGE --
FROM golang:1.25.1-alpine AS build-stage

# Install build dependencies
RUN apk add --no-cache upx

WORKDIR /app

# Download and Cache Go modules
COPY go.mod go.sum ./
RUN go mod download

# Copy the source code
COPY *.go .

# Build static binary with stripped debug info
RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o proj2

# Compress binary with UPX
RUN upx --best --lzma proj2

# -- RUNTIME STAGE --
FROM scratch

WORKDIR /app

# Copy binary from builder stage
COPY --from=build-stage /app/proj2 .

# Copy CA certificates for HTTPS
COPY --from=build-stage /etc/ssl/certs/ca-certificates.crt /etc/ssl
    /certs/

# Copy all input files to the image
COPY *.txt .

# Run default command
CMD ["./proj2"]
```

Figure 2.1: Dockerfile

This is mostly the same as Project 1, just removing some things dealing with `sqlite`, since we don't use that for this framework.

The build stage is where a chunk of the code and the data is. This data is only loaded once, as Go modules are cached to speed up rebuilds. It uses the

Go Alpine Image, which is small, lightweight, and efficient for our program. It installs build dependencies, like `upx`, which is a small image for usage in multi-stage Docker builds to compress binary files like Go or Rust [UPX Source17]. We set our working directory to `/app`, and download our program dependencies found in `go.mod`, and the checksum found in `go.sum`. After we get our source code into the Image, it will build the static binary (with some other extra settings to lower the size of the Image), and it will be compressed using `UPX`.

At the runtime stage, we start from an empty Image. This greatly reduces the Image size because the runtime build only contains what we explicitly copy. After copying the compressed static binary from the build stage, we have to worry about HTTP certificates. As explained by [CA Certificates24], we wouldn't be able to make the API call without a certification. Using a specific filepath found in [SSL Certificate Location on Unix/Linux09], we can get ourselves a certificate so we can use HTTP. After we get a certificate, this is where all of the user request test cases are added to the Image. This is why you would need to rebuild the Image if you wanted to add new test cases. Then, we specify the default command that will run when a container starts from this image.

The proj2 Docker Image size is only around 6.73 MB, mostly because the runtime uses scratch, but also because we are using a multi-stage build and requiring containing only the compiled binary, certificates, and input files, things we definitely need.

Of course, the images for Kafka, Prometheus, and Grafana are all much larger, but these were pre-made and there is nothing I could do to lower the sizes of those images. But, for the image that I could control, which is proj2, I made it as small as possible.

Most of the information I used to create my Dockerfile, for Golang specifically, was found here: [Golang Build Images Guide24].

2.2 Docker Compose

Here I will explain the fundamentals of the `docker-compose.yml` and another YML file that docker-compose uses, which is the `prometheus.yml` file. I will break down the `docker-compose.yml` file image by image. The first image I will talk about is Kafka, as shown below:

```yaml
  kafka:
    image: apache/kafka:latest
    container_name: kafka
    environment:
      - KAFKA_PROCESS_ROLES=broker,controller
      - KAFKA_NODE_ID=1
      - KAFKA_CONTROLLER_LISTENER_NAMES=CONTROLLER
      - KAFKA_LISTENERS=PLAINTEXT://0.0.0.0:9092,CONTROLLER
    ://0.0.0.0:29093
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092,
    CONTROLLER://kafka:29093
      - KAFKA_CONTROLLER_LISTENER_NAMES=CONTROLLER
      - KAFKA_CONTROLLER_QUORUM_VOTERS=1@kafka:29093
      - KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1
      - KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR=1
      - KAFKA_TRANSACTION_STATE_LOG_MIN_ISR=1
    ports:
      - "9092:9092"
    networks:
      - kafkanet
```

Figure 2.2: Kafka Service in docker-compose.yml

The first service listed is Kafka. The image comes from apache/kafka and the container name is kafka. The environment variables are all very important, but they allow the program to manipulate Broker image data [Apache/Kafka Data Used for docker-compose.yml File25]. My program uses this instead of setting up a separate Zookeeper image so that the total image size is smaller. It is also very important to look at the port number. The default port number for Kafka is 9092, and this is the port that is also specified in the my kafka section of my Go code. This port needs to be open in order for the Kafka readers and writers to work correctly. Also, something that you will notice is that all of these images are connected to the ”kafkanet” network. I named it ”kafaknet” because kafka was the first thing I configured to this network, and this whole framework is built around Kafka, so I thought it deserved the name (it also gave me the least trouble to program).

The second service I’ll explain is Prometheus, as shown below:

```yaml
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    ports:
      - "9090:9090"
    depends_on:
      - kafka
      - grafana
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    networks:
      - kafkanet
```

Figure 2.3: Prometheus Service in docker-compose.yml

The image that is obtained for prometheus is the prom/prometheus image, and the created container will be named prometheus. The default port for prometheus is 9090, which will be opened. This port is used for Prometheus to scrape the metrics it is sent [Setting up docker-compose.yml File for Prometheus20]. It also depends on the kafka and grafana services to be set up first. I think this isn't necessarily required, but it is just a safety thing, just in things things go wrong in running these services. The volume that is used to scrape these metrics is configured in the prometheus.yml file (that we will look at in a second), and it is also on the kafkanet network.

Before continuing with `docker-compose.yml`, I first want to explain what I included in my `prometheus.yml` file, as shown below:

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: "proj2"
    static_configs:
      - targets: ["proj2:8080"]
```

Figure 2.4: prometheus.yml

This file is used to tell Prometheus how often to scrape the metrics data [prometheus.yml Setup25]. It will scrape data every 5 seconds, and the target that Prometheus is configured to is proj2:8080. In the proj2 image that I will explain soon, the port that is opened is 8080. This port holds the metric data for Prometheus, which is why it needs to access it. When the program is running, the user can go to `http://localhost:8080/metrics` to look at the metrics.

Up next is the Grafana image in `docker-compose.yml`, as shown below:

```yaml
  grafana:
    image: grafana/grafana
    container_name: grafana
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
    ports:
      - "3000:3000"
    networks:
      - kafkanet
```

Figure 2.5: Grafana Service in docker-compose.yml

The image created is from grafana/grafana, and the container name will be grafana. The environment variables are important because it specifies the username and password the user needs to put in when going to `http://localhost:3000`. Grafana will ask you to change the password, but this isn't needed for the necessities of this program. The Grafana port number is 3000, and it is connected to kafkanet.

Finally, here is the proj2 image that I fully created myself (meaning it doesn't use a pre-built image like the others do), as shown below:

```yaml
  proj2:
    build: .
    container_name: proj2
    hostname: proj2
    environment:
      ##########
      # HERE IS WHERE YOU INPUT YOUR DATA
      API_KEY: YOUR_API_KEY
      # CAN OVERWRITE FILE AT RUNTIME USING -e FILE='filename.txt'
      FILE: inputX.txt
      WORKERS: 5
      # HERE IS WHERE THE ALERT THRESHOLDS ARE
      TEMP_LOW: 32
      TEMP_HIGH: 90
      HUMIDITY_LOW: 30
      HUMIDITY_HIGH: 70
      WIND_SPEED_HIGH: 40
      ##########
    ports:
      - "8080:8080"
    depends_on:
      - kafka
      - prometheus
      - grafana
    volumes:
      - prometheus:/data
    networks:
      - kafkanet
```

Figure 2.6: Proj2 Service in docker-compose.yml

The proj2 image is built from the Dockerfile explained previously. The container name and the hostname are both proj2. The hostname needs to be proj2 so that the running container doesn't have a strange name. The environment variables are were certain parameters are set. **This is where the OpenWeather API key is set**. The default file and the number of workers in the worker pool are also both set here. The environment variables also store the alert thresholds. This program has five different types of alerts, as displayed in the file. For example, if TEMP_LOW is set to 32, an alert will go off if the temperature for a given day in some ZIP code is below 32 degrees (all units are imperial for this program, meaning temperature is in Fahrenheit). The port opened is 8080, this way the metrics can be stored in proj2 so Prometheus can read them through `http://localhost:8080/metrics`.

This program is dependent on all of the other images, obviously. This service also creates a new volume which stores metric data for Prometheus. This volume is essentially the TSDB I will be using for this program. I had some trouble setting up the real TSDB with my program, so essentially I created a file which stores all metric data, which is persistent even when the program ends. Finally, it is connected to kafkanet.

2.3 Overall Program Flow

Below is a high-level overview of how the data in my program flows.

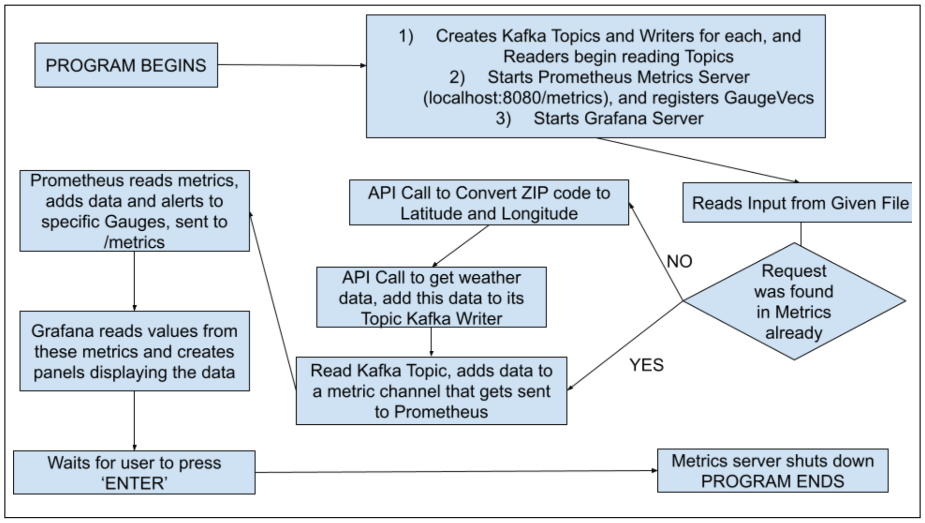
PROGRAM BEGINS
1) Creates Kafka Topics and Writers for each, and Readers begin reading Topics
2) Starts Prometheus Metrics Server (localhost:8080/metrics), and registers GaugeVecs
3) Starts Grafana Server
Reads Input from Given File
Request was found in Metrics already
NO
YES
API Call to Convert ZIP code to Latitude and Longitude
API Call to get weather data, add this data to its Topic Kafka Writer
Read Kafka Topic, adds data to a metric channel that gets sent to Prometheus
Prometheus reads metrics, adds data and alerts to specific Gauges, sent to /metrics
Grafana reads values from these metrics and creates panels displaying the data
Waits for user to press 'ENTER'
Metrics server shuts down PROGRAM ENDS

Figure 2.7: Program/Data Flow

I would like to explain how data flows throughout the program. The main way data is flowing is either through scraping or Go channels. Scraping is only used when getting data from Prometheus to Grafana, but channels is mainly how it flows, with some exceptions. Because I'm using goroutines, channels are very important to sent consistent data to other parts of my program [Channels].

The program starts off by initiating Kafka topics and writers for each topic. The program also begins reading each topic from the beginning of the file, this way data from previously ran programs can be found (we still need to use the Prometheus TSDB to find metrics though, since reading from Kafka logs without knowing the offset of that log is difficult). The metrics server begins, which can be found at `http://localhost:8080/metrics`, and the Gauges for each topic get registered with Prometheus. Also, Grafana is configured and started at `localhost:3000`.

After this, input from the given file is read. If that input was already found in the TSDB, that means we can skip a couple of steps and go straight to metric reading since this data will be read from the Kafka reader. If it wasn't found, two API calls will occur.

- The first API call will convert a given ZIP code to a latitude and longitude value. This is done using the GeoCoding API, which is part of the OpenWeather API, and it's free [Converting ZIP Codes to Long/Lat25]! This must be done since the second API requires longitude and latitude values
- The second API call will convert these longitude and latitude values into weather data over a span of the set number of days. All values use imperial units, as specified by the query sent to the API. The API being used is the "Call 5 day / 3 hour forecast data" [Getting Information25]. I am using this because it the only free option that OpenWeather gives that will be used for this framework. The limitation of this API is that it only limits to 5 days, so if the user inputs anything more than 5 days, it will be set to 5.

After the second API call, the forecast data will be written into Kafka using the writer for those specific topics. Now, no matter if the metric was found before or not, the Kafka reader will read the data for each topic and then pass it through the metrics channel, which get sent to Prometheus. The metric channel will read these metrics and add its data to the GaugeVecs (that was registered earlier) for the correct topic and create alerts if necessary. When these values are added to /metrics, Grafana will read them and create panels. Each dashboard represents a ZIP code, and each panel in each dashboard represents a topic. The alerts will also be displayed at the bottom of each dashboard.

These servers are up for the duration of the program, so in order to prevent the program from finishing, the program waits until the user presses enter.

2.4 Input Files

The three input files for this program are `inputX.txt`, `inputY.txt`, and `inputZ.txt`. Each serve some purpose to test this program. Of course, you can add your own test cases, but these are the ones I found helpful to test my program.
`inputX.txt`
Below displays the test case presented in `inputX.txt`

```
1 6|11552
2 "hello"|90210
3 3|90210
4 3|1234567890
5 2|12601
```

Figure 2.8: inputX.txt

This is the most basic test case, but it helps us look at the format that the files need to be in. Each line represents a users request, and each line has two fields, separated by |.

- The first field is the `days`, meaning how many days to forecast into the future.
- The second field is the `ZIP code`. It is important to note that the program only works with ZIP codes found in the United States because of how the OpenWeather API is set up

Of course here, "hello" is not a valid date (Line 2) and 1234567890 (Line 4) is not a valid ZIP code. The program will not end when it encounters these requests though, as it just skips them and moves on to the other requests.

`inputY.txt`
Below displays the test case presented in `inputY.txt`

```
3|94027
2|93108
1|99501
```

Figure 2.9: inputY.txt

I know I said each test case has its importance, but this is just a test case to see if everything is working. I added it to test if new dashboards get created when first running `inputX.txt`, then running `inputY.txt`. Since it is all new data, it will need to do API calls again, but it does create new metrics, and it also create separate dashboards in Grafana, which is what I wanted.

`inputZ.txt`
Below displays the test case presented in `inputZ.txt`

```
6|11552
"hello"|90210
3|90210
3|1234567890
2|12601
3|94027
2|93108
1|99501
5|10103
```

Figure 2.10: inputZ.txt

This test case combines request found in `inputX.txt` and `inputY.txt`, and it also contains some new requests. This makes sure that metrics from previous program iterations get overwritten by this new data, and also ensures more new data is created.

2.5 OpenWeather API

Now let's dive into the first couple of steps this program takes. Before doing any API calls, the program will parse each line of the valid, reading it and making sure the user input is valid.

This program uses wait groups in order to make sure everything syncs up correctly and the program doesn't finish before all components are finished. A waitgroup waits for multiple goroutines to finish [WaitGroups]. So, for example, a waitgroup is used with reading the files, making sure the program doesn't end until all lines of the files are read. Because this is happening concurrently with the main thread due to the goroutine being established, there is a chance the program could end earlier if wait groups weren't implemented as they are. On the next page, there is the code for reading each line of the given file using the Go scanner functionality (the Scanner can read the file, reading it line by line using the `Scan()` function [For File Reading and ZIP Code Collection25]).

```
435   // Reads file line by line concurrently (using goroutines and waitgroups)
436   for scanner.Scan() {
437     // Get text on current line
438     text := scanner.Text()
439
440     // Make a copy of the line number after its incrementation for better error messages
441     lineNumber++
442     currentLine := lineNumber
443
444     // Each of these goroutines work concurrently
445     fileWG.Go(func() {
446
447       // Validate the current request
448       req, success := parseLine(text, currentLine)
449
450       // If it is valid, send to precoordinate channel for further processing
451       if success {
452         preCoordinateChan <- req
453       }
454     })
455   }
```

Figure 2.11: Parsing the File (proj2.go)

The `parseLine()` function (Line 448) will return an object created from a custom structure named PreCoordinateRequest. This structure handles all of the data that the API must know when converting a ZIP code into latitude and longitude. Below lists the members of this structure:

```
// A structure based off of the user input (BEFORE converting ZIP code to coordinates)
type PreCoordinateRequest struct {
  Days    int
  ZIPCode string

  LineNum int
}
```

Figure 2.12: PreCoordinate Request Struct (proj2.go)

This structure holds the number of days the user requested to forecast, the ZIP code they put in the request, and also stores the line number for debugging purposes.

This request is put into the preCoordinateChan, which is a Go channel that stores any incoming pre coordinate request (pre-coordinate meaning before the request got converted into coordinates). Another worker pool, which deals with this channel, will receive this channel data whenever a request is made, and it will convert the ZIP code to coordinates, if it wasn't found in the TSDB, as shown below:

```go
// Goroutine that collects data from the preCoordinate channel
// Worker pool created for parallel GeoCoding API Requests
for range numWorkers {
  zipCodeWG.Go(func() {
    // Will wait until data gets put into the requests channel
    for req := range preCoordinateChan {

      // Will check if this request already has results
      exists := isInTSDB(req)

      // If not in Prometheus TSDB, must create a new request and call API
      if !exists {
        // Convert ZIP code to coordinates, then add to request channel
        newRequest, success := convertToCoordinates(req, key)
        if success {
          requestsChan <- newRequest
        }
      }
    }
  })
}
```

Figure 2.13: Converting to Coordinates (proj2.go)

As you can see, if the request doesn't exist, it will need to do the GeoCoding API call in the `convertToCoordinates` function. This API call converts a ZIP code to latitude and longitude [Converting ZIP Codes to Long/Lat25]. The format of this API call found in `convertToCoordinates()` is seen below:

```go
// Make API request to get coordinates (assuming UNITED STATES)
url := fmt.Sprintf("http://api.openweathermap.org/geo/1.0/zip?zip=%s,US&appid=%s", zipCode, key)

// Make a HTTP GET request to this URL, returning an HTTP response
resp, err := http.Get(url)
```

Figure 2.14: API Call to Convert to Coordinates (proj2.go)

Again, this is all skipped if it was found in metrics file. If this request was successful, the `convertToCoordinates()` function will return an object from a new structure named `PostLocationRequest`. Below lists the members of this structure:

```go
// A structure based off of the user input (AFTER converting ZIP
    code to coordinates)
type PostLocationRequest struct {
  Days    int
  Name    string
  Lat     float32
  Lon     float32
  ZIPCode string

  LineNum int
}
```

Figure 2.15: PostLocation Request Struct (proj2.go)

The number of days, the ZIP code, and the line number are all the same from the `PreCoordinateRequest`, but the name, latitude, and longitude are all values that were returned from the API. These values were decoded from the JSON API response and put into the returned object. This object is then passed into the `requestsChan`, and now the data gets passed to another goroutine in another worker pool, as shown below:

```go
  // Goroutine that collects data from the request channel
  // Worker pool created for parallel API Requests
  for range numWorkers {
    resultsWG.Go(func() {
      // Will wait until data gets put into the requests channel
      for req := range requestsChan {
        processRequest(req, key, kafkaWriters)
      }
    })
  }
```

Figure 2.16: Processing the Request (proj2.go)

When data is put into the requests channel, this goroutine will read that data and process the request, doing the actual API call to return weather forecast data for specified amount of days the user put. Below is what the actual API call looks like:

```go
  // Make API request to get results (using imperial units)
  url := fmt.Sprintf("https://api.openweathermap.org/data/2.5/
    forecast?lat=%f&lon=%f&cnt=%d&units=imperial&appid=%s", lat,
    lon, cnt, key)

  // Make a HTTP GET request to this URL, returning an HTTP
    response
  resp, err := http.Get(url)
```

Figure 2.17: API Call to Get Weather Data (proj2.go)

The data used for this API call are the latitude, longitude, the number of data to return, the units the results will be in (imperial), and the API key [Getting Information25]. Because this API returns data at three hour increments, and we only want one dataset per day, the `cnt` variable is the number of days specified multiplied by 8 (because 8 * 3 hours = 24 hours = 1 day).

When the API responds with many results, they are decoded into many different custom structures that I created for the sole purpose of gathering these results into a neat and organized way. I'm not going to go deep into my creation of these structures, as they are just based on the JSON results.

After the API calls are complete, the organized data is written into Kafka topics, as I will explain in the next section.

2.6 Kafka

Before explaining how the API results get passed into Kafka writers, I first want to explain how the Kafka server starts and how the Kafka readers start reading data from the Kafka logs. When the program begins, it will begin to initialize the Kafka writers. Before initializing these writers however, the program needs to make sure Kafka is up and running. This is what the `waitForKafka()` function is used for, as shown below:

```go
// Waits for Kafka to be set up
func waitForKafka() {
  retryDelay := 2 * time.Second

  // Once Kafka is officially setup and this connection is
    successful, the function will finish
  for {
    conn, err := kafka.Dial("tcp", brokerPort)

    if err == nil {
      conn.Close()
      return
    }
    fmt.Println("Kafka is not ready yet. Retrying...")
    time.Sleep(retryDelay)
  }
}
```

Figure 2.18: Waits for Kafka to Start (kafka.go)

This function will wait for a successful connection to occur with `kafka:9092`, which is what the Broker uses to connect [Kafka Connections25]. It keeps retrying every two seconds until it eventually connects. If everything was set up correctly with the Docker containers, Kafka may take at most one try to connect initially.

After a connection to Kafka is established, the Kafka writers will be created. These writers are used to publish data to its set topic [Kafka Writers25]. For this program, there is one topic per writer. So, there are a total of four writers:

- tWriter: Writer for the temperature topic (includes temperature and feelslike data)
- hWriter: Writer for the humidity topic (includes humidity data)
- wWriter: Writer for the wind topic (includes wind speed and wind degrees data)
- cWriter: Writer for the cloud topic (includes cloud coverage data)

These topics are grouped together into my custom based `KafkaWriters` structure in order to make it neat and organized. Also, these writers are closed at the end of the program.

Now we need to ensure the topics the readers are reading from exist before they start reading. This is where the `ensureKafkaTopic()` function is used. The function gets passed a topic, and then it makes sure that the topic exists by reading its partitions and seeing if it exists. If it doesn't exist, then Kafka must create it using Controllers, since you need a Leader connection to create topics in Kafka [Starting Kafka and Creating Topics20]. In Kafka, the active controller is the leader of the metadata topic's single partition and it will receive all writes [Controllers to Create Topics25].

```go
// First, find the Kafka controller (responsible for topic
  creation)
// In Kafka, only the controller broker can create topics
controller, err := conn.Controller()
check(err)

// Connect to the Kafka controller
controllerConn, err := kafka.Dial("tcp", fmt.Sprintf("%s:%d",
  controller.Host, controller.Port))
check(err)
defer controllerConn.Close()

// Define topic configuration: 1 partition, 1 replica
topicConfigs := []kafka.TopicConfig{
  {
    Topic:             topic,
    NumPartitions:     1,
    ReplicationFactor: 1,
  },
}

// Send request to Kafka controller to create the topic
err = controllerConn.CreateTopics(topicConfigs...)
check(err)
```

Figure 2.19: Creating Kafka Topics (kafka.go)

Now we can start reading these topics. The Kafka readers are used to read Writer data from Kafka logs [Reading Kafka Data25]. A worker pool will consume Kafka topic data whenever it reads a new entry from the log. Below is how the reader is set up.

```go
190 // Reads messages that come through topics
191 func consumeKafkaTopic(ctx context.Context, topic string) {
192
193   // Creates a new Kafka reader to read data coming from this topic
194   reader := kafka.NewReader(kafka.ReaderConfig{
195     Brokers:     []string{brokerPort},
196     Topic:       topic,
197     StartOffset: kafka.FirstOffset,
198     MaxWait:     100 * time.Millisecond,
199   })
200   defer reader.Close()
```

Figure 2.20: Creating Kafka Reader (kafka.go)

The Kafka readers will always start reading from the beginning of the log (Line 197) in order to make sure that data from previous program iterations are also read. This makes it easier so we don't have to keep pulling from the Prometheus metric data, instead it is automatic through Kafka readings. The context is used to tell the Reader to STOP when the program ends.

When the Reader for that topic retrieves the data, it will unmarshal it into a `WeatherMessage` structure. Below lists the members that make up this structure.

```go
// Structure that holds the consumer data that will be sent to
    Prometheus
type WeatherMessage struct {
  Topic       string
  Zip         string
  Date        string
  Temperature float64 ‘json:"Temp"‘
  FeelsLike   float64 ‘json:"FeelsLike"‘
  Humidity    float64 ‘json:"Humidity"‘
  WindSpeed   float64 ‘json:"Speed"‘
  WindDegree  float64 ‘json:"Degree"‘
  Cloud       float64 ‘json:"CloudPercent"‘
}
```

Figure 2.21: WeatherMessage Structure (kafka.go)

Because there exists a Kafka reader for each topic, not all of these fields will be filled out. The Topic, ZIP, and Date will be manually filled out by the program, and only one of the topics will be filled out when the data is unmarshaled. After the structure is created, it gets passed to Prometheus using the metrics channel.

Before moving on to Prometheus, I still haven't explained by the data gets written in the first place! If you recall, when the WeatherMap API call occurs, the data gets decoded into many different structures that stem from the API response. Now, when the response data is gathered, the program creates many different payloads that it will write into Kafka. Here is an example of a payload being created for temperature.

```go
    // Create metric-specific payloads to add to Kafka Writers
    tempPayload := TemperaturePayload{
      Location:  location,
      Date:      date,
      Temp:      float64(r.Main.Temp),
      FeelsLike: float64(r.Main.FeelsLike),
    }
```

Figure 2.22: Creating Payload for Temperature (proj2.go)

This payload contains the data that Kafka (and indirectly Prometheus and Grafana) needs to know about the results retrieved from the API. Now, these payloads need to be published to the correct Kafka writer. This is shown below for temperature, but it is done for every topic:

```go
    // Key for each payload is the ZIP code and the date (zipcode-date)
    key := fmt.Sprintf("%s-%s", zipCode, date)

    // Publish payloads to their specific Kafka writer topics
    tempBytes, _ := json.Marshal(tempPayload)
    kWriters.TempWriter.WriteMessages(context.Background(), kafka.Message{Key: []byte(key), Value: tempBytes})
```

Figure 2.23: Publishing Payload for Temperature (proj2.go)

First, these payloads must be marshaled into bytes because thats what Kafka writers accept as input. Then, a message is written to the writer, with the key being the ZIP code date pair, and the value being the marshaled bytes that was just created. These will essentially write the messaged payloads to Kafka, allowing the readers to actual read the API data [Writing Message Payloads to Kafka25].

2.7 Prometheus

Now I will move on to the Prometheus portion of this framework. The first thing to talk about is setting up the Prometheus metrics server. This displays the metric data on `http://localhost:8080/metrics`. When the program begins, it will open up this server in a separate goroutine called `startMetrics()`, as shown below.

```go
// Starts the HTTP server for Prometheus (avaliable at localhost
    :8080/metrics)
func startMetrics() {
  http.Handle("/metrics", promhttp.Handler())
  if err := http.ListenAndServe(":8080", nil); err != nil {
    fmt.Println("Prometheus HTTP server failed:", err)
    os.Exit(1)
  }
}
```

Figure 2.24: Starting Metrics Server (prometheus.go)

The reason why this is started in a goroutine is because the `http.ListenAndServe` function that looks at port 8080 will never finish. But, this is the function that starts that metric server.

In Golang, before the `main()` function is called, the `init()` function is called to initialize anything that needs initialization [Understanding init in Go19]. In this case, the Prometheus Gauges need to be registered. Below is an example of the temperature gauge being created and then being registered in the `init()` function.

```go
  // PROMETHEUS GAUGES FOR EACH TOPIC
  tempGauge = prometheus.NewGaugeVec(
    prometheus.GaugeOpts{
      Name: "temperature",
      Help: tempHelp,
    },
    []string{"location", "date"},
  )
```

Figure 2.25: Creating Temperature GaugeVec (prometheus.go)

```go
// Ran before main()
func init() {
  // Register metrics with the default registry safely
  safeRegister(tempGauge, "temperature")
```

Figure 2.26: Registering Temperature GaugeVec (prometheus.go)

The temperature gauge is created with a name and some help data [Creating Gauge Data for Prometheus25]. I set this earlier as a string just explaining the topic. The key for this metric is the ZIP code and date combination. It is important to note that alert gauges are also made for each alert.

The `safeRegister()` function is one that I created to ensure that Gauges don't get registered multiple times. This ensures that a Gauge is registered once. A

Gauge is registered using the prometheus function `mustRegister()` [Registering Gauges for Prometheus25].

This `init()` function also makes sure the environment variables passed for the alerts are all valid. If they aren't valid, the alert thresholds get set to default values.

Before explaining how the metrics channel data is used to set values to these Gauges, I would like to explain how previous data is found in the TSDB or metric data volume. Before doing any API calls, the program needs to figure out if it already is holding its data. It does this by checking the TSDB, which is the metrics volume created. It reads the file and if it sees the same key values as found in the request, that means no API calls need to be done.

```go
// Reads this file
scanner := bufio.NewScanner(file)
for scanner.Scan() {
  var msg WeatherMessage

  // Each line will be converted to a msg structure
  err := json.Unmarshal(scanner.Bytes(), &msg)
  if err != nil {
    continue
  }

  // If the same values are found as the request, then that means
   the API does NOT need to be called anymore
  if msg.Zip == zip && msg.Date == date {
    fmt.Printf("Found metric for %s-%s in file\n", zip, date)
    return true
  }
}

return false
```

Figure 2.27: isInTSDB() File Reading (prometheus.go)

Now, whenever data in the metrics channel is read (data was written to the channel using the Kafka reader), the `updateMetrics()` function is called to update the specific Gauge for that topic. Here is an update of the use of this function when the data passed in is temperature data.

```
183  // Update Gauges with metric data from Kafka for EACH topic
184  // Also sets alert gauges if necessary
185  switch msg.Topic {
186  case "temperature":
187    tempGauge.WithLabelValues(msg.Zip, msg.Date).Set(msg.
       Temperature)
188    feelsLikeGauge.WithLabelValues(msg.Zip, msg.Date).Set(msg.
       FeelsLike)
189
190    // Set alert gauge to 1 or 0 depending on temperature
191    if msg.Temperature > tempHigh {
192      alertTempHigh.WithLabelValues(msg.Zip, msg.Date).Set(1)
193    } else {
194      alertTempHigh.WithLabelValues(msg.Zip, msg.Date).Set(0)
195    }
196
197    if msg.Temperature < tempLow {
198      alertTempLow.WithLabelValues(msg.Zip, msg.Date).Set(1)
199    } else {
200      alertTempLow.WithLabelValues(msg.Zip, msg.Date).Set(0)
201    }
```

Figure 2.28: Setting Gauges values (prometheus.go)

So the key for any topic is the ZIP-date combination like the Kafka writers, and the value is going to be the data that we want displayed (Line 187). In the case of temperature, it also has "feelslike" data we want to display, so that Gauge will be updated as well. Temperature also has alerts, so it will compare that temperature to these alerts to see if we should set the value to 1 or keep it at 0 (Lines 191-201). If the alert is set to 1, that means the alert is active and the user should be warned, but this will all be done in Grafana. After this data is read and set to these Gauges, it will be written to the metric data TSDB, needing to marshal this data into bytes to read this file [Marshaling to Write to the File25].

2.8 Grafana

The final puzzle piece for this Framework to explain is Grafana. First of all, at the beginning of this program, the program waits until Grafana has started up before doing anything. This is very similar to the `waitForKafka()` function described earlier. Basically, this function does some API calls to the /api/health endpoint for Grafana to see if its up and running on `http://localhost:3000` by doing some HTTP requests [Health API25]. The program will retry every two seconds, and it will timeout after a minute, but this shouldn't be any issue when starting the program regularly. It should be setup on the first try.

Now, Grafana doesn't do anything until all of the API calls, Kafka writing and reading, and Prometheus metrics are all done being configured. So, at the end of the program, Grafana is properly setup. The first thing that needs to be done is to connect Grafana to Prometheus in the `setupPrometheusDataSource()` function. Below is how this function is implemented:

```go
// Ensures Grafana has Prometheus configured as a data source
func setupPrometheusDataSource() {
  client := &http.Client{}

  // Define the Prometheus data source payload
  // The URL is the Prometheus container URL
  dataSource := map[string]any{
    "name":      "Prometheus",
    "type":      "prometheus",
    "url":       "http://prometheus:9090",
    "access":    "proxy",
    "isDefault": true,
  }

  // Marshal the dataSource map into JSON for the HTTP request body
  payload, _ := json.Marshal(dataSource)

  // POST /api/datasources
  req, _ := http.NewRequest("POST", grafanaURL+"/api/datasources",
    bytes.NewBuffer(payload))
  req.SetBasicAuth(grafanaUser, grafanaPass)
  req.Header.Set("Content-Type", "application/json")

  // Sends the request
  resp, err := client.Do(req)
  if err != nil {
    fmt.Println("Error creating Prometheus data source:", err)
    return
  }
  defer resp.Body.Close()

  // If the request status is successful, that means Prometheus was
     configured successfully
  if resp.StatusCode >= 200 && resp.StatusCode < 300 {
    fmt.Println("Prometheus data source configured successfully!")
  }
}
```

Figure 2.29: Connecting Prometheus to Grafana (grafana.go)

Like how we waited for Grafana to start, we again are doing some API calls, but this time to the api/datasources directory and setting up some basic authentication to create the Prometheus data source [Data Source for Connecting Prometheus to Grafana25].

Now Prometheus scraping will be useful because Grafana can read this data and we can use it to create dashboards! The scraping interval was set in `prometheus.yml`, which was explained earlier. When the program is running, you can actually see how often it is scraping from the metrics data using `http://localhost:9090/targets`.

Now it is time to create the dashboards. A dashboard is creating for each ZIP code. Each panel in each dashboard represents a topic. The list of all ZIP codes was found reading all unique ZIP codes from the metrics file in the function `getAllZipCodes()`. The panels are created by manipulating the set Grafana JSON Panel code [How Grafana Dashboards are Structured in JSON25]. Using this JSON, we can add the Prometheus data to the expressions [Prometheus Metrics in Grafana25], as shown below:

```go
// Create graphs for each topic
for i, topic := range metricTopics {

  // Grafana JSON Panel code that gets manipulated to add the data that is needed
  panel := map[string]any{
    "type":  "graph",
    "title": namedTopics[i],
    "id":    panelID,
    "gridPos": map[string]any{
      "h": 8,
      "w": 24,
      "x": 0,
      "y": yPos,
    },
    // The targets will get the data we need
    "targets": []map[string]any{
      {
        // Get last value over 15s window for this ZIP and metric
        "expr":         fmt.Sprintf("last_over_time(%s{location=\"%s\"}[15s])", topic, zip),
        "legendFormat": "{{date}}",
        "refId":        "A",
      },
    },
```

Figure 2.30: Manipulating Panel JSON (grafana.go)

There is more JSON that gets changed, but the targets ”expr” is the most important part, since this is where the data is written for each topic. Below is a display on how the temperature panel may look like:

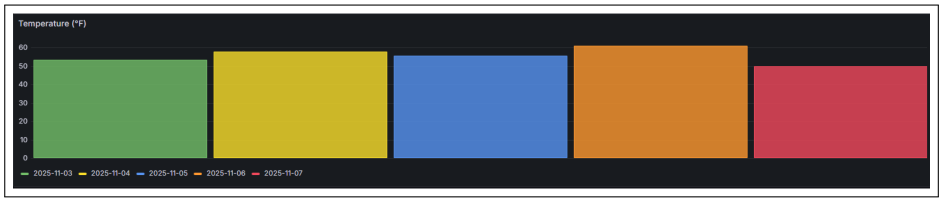

Figure 2.31: Temperature Panel

It is hard to see the actual data in the image, but the general picture is there. For a given ZIP code, each color represents a date, and the Y axis is the temperature in Fahrenheit. The data is in a bar graph format, but it is using the time-series data from Prometheus [Using Time Series25].

After all panels are created for the given ZIP code, alert panels are also created. These panels will say "ALL GOOD!" if there was no alert, but it will give the alert day if there was one. Here is an example of an alert below:

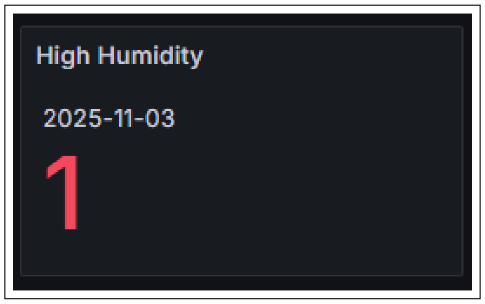

Figure 2.32: Alert Example

The above example shows that there is high humidity on November 3rd for this ZIP code. All other alerts also display, but their panels say "ALL GOOD!".

Each dashboard creates their own panels and alerts, and at the end these dashboards are pushed to Grafana so that they can display the given data.

3 Conclusion

3.1 One Small Issue

There is one small issue that I spent hours debugging, and I got it down to something extremely small, but still somewhat of an issue nonetheless.

Sometimes, very rarely, for a couple of frames (I'm not even kidding), the data displayed in Grafana will sometimes double. This can be easily fixed with a quick refresh of the page, but this was an issue I've been trying to fix.

The main reason why this happens is because the field data reducer is set to the sum of metric data received by Grafana. It should be set to 'last' and not 'sum', but I couldn't figure out a way to automate this in my Go code. If you really want to, you can go into the JSON Panel Configuration and change each transformation reducer from 'sum' to 'last', but it just seems like a lot of work for something that can be fixed as easily as a refresh.

3.2 Use of AI

I am proud to say that I didn't use any AI to generate any of the code in this program. I spent the first couple of weeks studying how Kafka, Prometheus, and Grafana all work before actually building up my program. However, this doesn't mean I didn't use AI at all.

I was having trouble overriding the JSON panel data with my own data in my Go code for Grafana. So, I did use some OpenAI ChatGPT Version 4. The main prompt I used was "Get the JSON Panel data in Grafana for a given time series". This returned the JSON format that I needed to use in order to override the default set Grafana JSON.

Other than that, I Googled everything else (as you can see by how many references I have) to try to learn it myself.

3.3 Conclusion / Future Work

Overall this was a major learning experience for me, especially when dealing with cloud computing software principles. This might have been the hardest thing I've worked on in this field, but I really learned so much from it. I knew a lot about Kafka and Prometheus from class, but actually using it in a practical setting made me understand it so much more.

Here is some things I would change in the future about my code for this project:

- API calls may still happen if ZIP code was wrong since it wasn't found in the metrics table. Maybe I could create a new volume that stores wrong ZIP data, therefore the GeoCoding API call doesn't need to happen again.
- Removing duplicates from the metrics data file was something I tried to work on but it seemed very difficult to do. This is definitely something I would improve, but for the small scale of this project, this doesn't really affect the file size or runtime performance much.
- Organize the code just a little bit better, as it can be confusing without explicit documentation.

Below are my references for this Project.

References

[Apache/Kafka Data Used for docker-compose.yml File25] Docker Hub. *apache/kafka Docker Image*. 2025. URL: https://hub.docker.com/r/apache/kafka.

[CA Certificates24] Docker Documentation. *CA Certificates*. 2024. URL: https://docs.docker.com/engine/network/ca-certs.

[Channels] Mark McGranaghan. *Channels*. URL: https://gobyexample.com/channels.

[Controllers to Create Topics25] Confluent. *Controllers to Create Topics*. 2025. URL: https://developer.confluent.io/courses/architecture/control-plane.

[Converting ZIP Codes to Long/Lat25] OpenWeatherMap. *Geocoding API*. 2025. URL: https://openweathermap.org/api/geocoding-api.

[Creating Gauge Data for Prometheus25] Prometheus Client Go. *Gauge Documentation*. 2025. URL: https://pkg.go.dev/github.com/prometheus/client_golang/prometheus#Gauge.

[Data Source for Connecting Prometheus to Grafana25] Grafana. *HTTP API - Data Source*. 2025. URL: https://grafana.com/docs/grafana/latest/developers/http_api/data_source.

[For File Reading and ZIP Code Collection25]	Go Lang. *bufio Scanner Documentation*. 2025. URL: https://pkg.go.dev/bufio#Scanner.
[Getting Information25]	OpenWeatherMap. *5 Day / 3 Hour Forecast API*. 2025. URL: https://openweathermap.org/forecast5#5days.
[Golang Build Images Guide24]	Docker Documentation. *Golang Build Images Guide*. 2024. URL: https://docs.docker.com/guides/golang/build-images.
[Health API25]	Grafana. *HTTP API - Other*. 2025. URL: https://grafana.com/docs/grafana/latest/developers/http_api/other.
[How Grafana Dashboards are Structured in JSON25]	Grafana. *HTTP API - Dashboard*. 2025. URL: https://grafana.com/docs/grafana/latest/developers/http_api/dashboard.
[Kafka Connections25]	Segment.io. *kafka-go Documentation*. 2025. URL: https://pkg.go.dev/github.com/segmentio/kafka-go#section-readme.
[Kafka Writers25]	Segment.io. *kafka-go Writer Documentation*. 2025. URL: https://pkg.go.dev/github.com/segmentio/kafka-go#readme-writer-godoc.

[Marshaling to Write to the File25]	Golang Cafe. *Golang JSON Marshal Example*. 2025. URL: https://golang.cafe/blog/golang-json-marshal-example.html?utm_source=chatgpt.com.
[Prometheus Metrics in Grafana25]	Grafana. *Prometheus Metrics in Grafana*. 2025. URL: https://grafana.com/docs/grafana/latest/datasources/prometheus.
[prometheus.yml Setup25]	Prometheus. *Configuration: prometheus.yml Setup*. 2025. URL: https://prometheus.io/docs/prometheus/latest/configuration/configuration.
[Reading Kafka Data25]	Segment.io. *NewReader in kafka-go*. 2025. URL: https://pkg.go.dev/github.com/segmentio/kafka-go#NewReader.
[Registering Gauges for Prometheus25]	Prometheus Client Go. *Prometheus Package Documentation*. 2025. URL: https://pkg.go.dev/github.com/prometheus/client_golang/prometheus.
[Setting up docker-compose.yml File for Prometheus20]	Ulises. *Simple Prometheus Setup on Docker Compose*. 2020. URL: https://mxulises.medium.com/simple-prometheus-setup-on-docker-compose-f702d5f98579.
[SSL Certificate Location on Unix/Linux09]	ServerFault. *SSL Certificate Location on Unix/Linux*. 2009.

URL: https://serverfault.com/questions/62496/ssl-certificate-location-on-unix-linux.

[Starting Kafka and Creating Topics20]	Stack Overflow. *How to Create Kafka Topic Using segmentio's kafka-go*. 2020. URL: https://stackoverflow.com/questions/61618623/how-to-create-kafka-topic-using-segmentios-kafka-go?.
[Understanding init in Go19]	Gopher Guides. *Understanding init in Go*. 2019. URL: https://www.digitalocean.com/community/tutorials/understanding-init-in-go.
[UPX Source17]	gruebel. *UPX Source*. 2017. URL: https://github.com/gruebel/docker-upx.
[Using Time Series25]	Grafana. *Time Series Visualization*. 2025. URL: https://grafana.com/docs/grafana/latest/panels-visualizations/visualizations/time-series.
[WaitGroups]	Mark McGranaghan. *WaitGroups*. URL: https://gobyexample.com/waitgroups.
[Writing Message Payloads to Kafka25]	Segment.io. *Writer.WriteMessages*. 2025. URL: https://pkg.go.dev/github.com/segmentio/kafka-go?#Writer.WriteMessages.