CYBR410 Group Project: Big Deployment

Paul Aguilar *Front-End*
Nicholas Burlakov *Front-End*
Nehemiah Fiedls *Logging*
Eric Leachman *Orchestration*
Thomas Longwell *Defense*
Robert Rutherford *Defense*
Spencer West *Services*

Abstract

The goal of this project is to work as a group with each member implementing a different aspect making up a larger deployment.

1 Introduction

This project is designed to have groupmates work together to design and implement a full deployment. Each teamate is assigned a specific piece of the deployment as their responisbility adn each section of this paper relates to a section of the deployment. Front-End is in charge of creating the user interface of the deployment, the web server, this will host the services which for this project

Rewrite In today's digital landscape, deploying web services with security and efficiency is paramount. This project focuses on harnessing the power of Kubernetes (K8s) orchestration to achieve these goals. Kubernetes provides an ideal platform for hosting critical web services. In this endeavor, we will deploy an Nginx server to deliver an IP-to-Physical Address service and a weather service, ensuring these services are highly available and performant.

Security is a cornerstone of our approach. We will implement a multi-layered defense strategy, incorporating honeypot mechanisms to detect and deflect potential threats. For real-time monitoring and comprehensive logging, we will leverage Sysdig, a powerful tool that offers deep visibility into the system's behavior. This combination of advanced security measures and monitoring tools will safeguard our services against a wide range of cyber threats.

Furthermore, by utilizing Kubernetes, we aim to optimize resource utilization and enhance fault tolerance. Kubernetes' inherent capabilities in managing containerized applications will allow us to seamlessly scale our services according to demand, ensuring continuous availability and reliability. This project underscores the critical importance of efficient and secure web service deployment in meeting the growing demands of modern users and enterprises.

2 Orchestration

This is the orchestration section (Eric)

3 Front end

The front end is designed around a few simple HTML template pages, stylized with a CSS style guide. There are three main templates and a variation on two – the first template is the index, a home page that is repeatedly linked on some blank links in the navigation bar. There are also two pages dedicated to the address and weather API's. These pages have two versions, a search and an output template. The search template uses a simple form for the user to input a URL. The output template displays the results from the API call. These files are hosted via a flask server.

The flask server uses the render_template and request objects to function seamlessly with the HTML and CSS. Following traditional directory organizations, a templates and static directory are used to hold those files for flask to access.

```
/app
     - fServer.py
     /templates
        - index.html
     /static
logo png's
/fonts
       /styles
             - style.css
```

Flask's render_template is used to display the correct HTML template in any given scenario, either from the hostWeather method or directly in the HTML links. The request object is used to handle the form submissions so that the user provided URL's can be used by the API methods to generate the correct output. Both hostAddress and hostWeather use a similar method to pass the URL correctly depending on whether a POST request was made from a form submission, via requests request.form method or directly from the URL

query string. From there the API methods are used to gather the data, which is the outputted using the render_template method and displayed with in the output template.

Listing 1: hostWeather method handles GET/POST requests

```
@app.route("/weather", methods=['GET', 'POST'])
def hostWeather():
    if request.method == 'POST':
        # Logic to handle form submission via POST request
        incomingHost = request.form['url']
    else:
        # Logic to handle GET request
        incomingHost = request.args.get('url', '')

    ipAddress = getIP(incomingHost)

    if checkCache(cacheW, incomingHost):
        weather = cacheW[incomingHost]
    else:
        weather=getWeather(incomingHost)
        addCache(cacheW, incomingHost, weather)
```

While the host methods route to the correct output page, the remaining routes direct a user to the index or the search pages. The following sequence diagram shows the basic user interaction with the site:

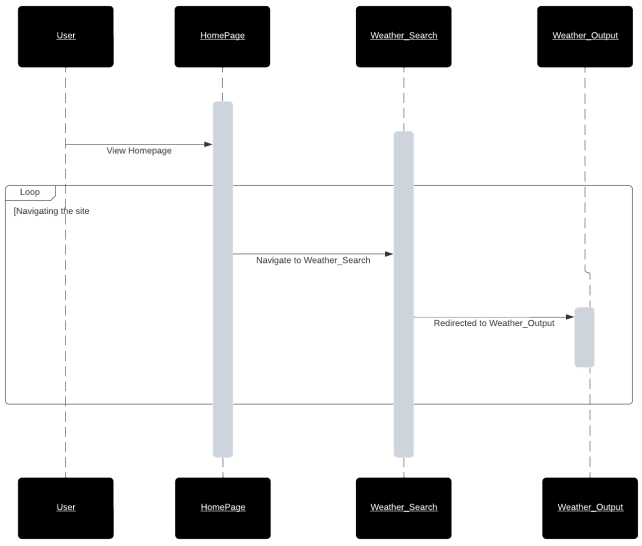

Figure 1: Environment Diagram

4 Services

This web application incorporates two API's in the flask server. One service takes a URL and outputs the address of the registered owner based on the whois registry. The other uses that address and another weather API to output the weather of that given location. It also uses a list to cache the data of visited sites, to improve performance. To do all this, the API is broken up in to multiple methods, which ultimately called by the hostWeather and hostAddress methods when a form submission is made on the webpage.

The first method that is used is getIP, which first strips the incoming URL of any whitespace and then uses the socket library's gethostbyname method to save the IP. this method is then used by both getWeather and getAddress in order to find the related information. getAddress uses the subprocess' getstatusoutput method to run the whois commandline tool to search for the registrar information based on the provided IP. It then splits the incoming tuble by a newline and parses out the different address fields, combines them and returns a whole address.

The getWeather method then passes that address in to a geocoding API to find the latitude and longitude of the address, which is required by the weather API that is used. The geocoding call returns a json file, that needs to be parsed and split between x and y coordinates correctly. The request's get method is then used to make an API call to weather.gov via the command line again. This outputs another json file that is then parsed to find the correct URL that contains the weather information. The get method is used once again on that forecast URL, which outputs the final json file, this is correctly parsed and returned as the weather for the given IP.

Listing 2: getWeather method

```
def getWeather(incoming):


    ipAddress= getIP(incoming)

    addy = getAddress(ipAddress)

    geocodingApi="https://geocoding.geo.census.gov/geocoder/locations/address?street="+addy[0]+"&city="+addy[1]+"&state="+addy[2]+"&zip="+addy[3]+"&benchmark=Public_AR_Current&format=json"
    georesponse=requests.get(geocodingApi)
    js = json.loads(georesponse.text)

    outputx=js['result']['addressMatches'][0]['coordinates']['x']
```

```
    outputy=js['result']['addressMatches
        '][0]['coordinates']['y']
    lat=format(outputy)
    lon=format(outputx)


    # base API string for weather.gov
    weather_s = "https://api.weather.gov
        /points/"

    # use the commandline input and the
        weather_s to make API call
    response = requests.get(weather_s+
        lat+","+lon)

    # convert it to json
    js = json.loads(response.text)

    # find the forecast URL based on the
         API page
    forecast_URL = js['properties']['
        forecast']

    # call the API again to get
        theforecast
    final_response = requests.get(
        forecast_URL)

    #parse json
    js = json.loads(final_response.text)

    #print the forecast
    weather=(js['properties']['periods'
        ][0]['detailedForecast'])

    return(weather)
```

The remaining methods handle the caching, either directly adding to it based on the correct list or traversing it to search for the given key which returns true or false. The host methods use this to first check the cache and then output the value based on the given key or add the key/value pair to the given cache if it isn't found. To summarize, this script creates a Flask web server that provides endpoints for IP address lookup and weather information retrieval, caching the results to improve performance. It also demonstrates how to integrate external APIs and system commands into a Flask application.

5 Logging

The bash script I've made automates the monitoring of a system by capturing system activity using the sysdig tool every 30 minutes. Each session stops when the data file reaches about 20 MB or after 4 hours of total runtime. The script checks if each generated file exceeds a size limit of 20MB and, if so, runs a specific program. This can be useful for system monitoring, security analysis, and troubleshooting. The script helps manage data capture efficiently but requires careful consideration regarding storage, privacy, and the potential impact on system performance.

6 Defense

For the defense of our system we implemented multiple defensive techniques. The first defensive technique is built into our system design and is the load balancer which functions as moving target defense by putting users into one version or

References