Stephen Haugland and Griffen Marler
Scott Griffith
CS 313
9 December 2019

Discussion of Work and Analysis: SlackWeather

Discussion of Work:

For our Networks final project, we decided to work with different API's and RDP through Microsoft Azure to further deepen our knowledge and personal growth as Networks students and programmers. When we initially started our final project, we were aiming to create a Twitter bot that queried a Twitter user's tweets and analyzed 'X' number of tweets through a sentimental analysis API. We would then be able to report on whether or not a Twitter user is a consistently negative, positive, or neutral tweeter through our application. We really liked this idea and tried extremely hard to implement it. We applied and were approved for Twitter authentication tokens at https://developer.twitter.com/, successfully built and compiled the twitcurl library on one of our machines using the following links (https://github.com/swatkat/twitcurl and https://www.youtube.com/watch?v=eITwklZMckc), and began the process of querying tweets. After successfully querying tweets and attempting to share our first progress from one computer to another via USB transfer we found that the build configuration required for the Twitcurl was extremely unportable, and would take a significant amount of time per machine to transport. Because of this, we leaned toward a more portable and open source approach. Upon thinking about social platforms similar to Twitter, we stumbled across a C++ library for the workspace application "Slack". This library called "Slacking" (https://github.com/coin-au-carre/slacking) gave us the ability to implement a bot in a slack

workspace of our choosing. This library directly appealed to us as it was a “header only” library that also came included with a JSON parsing library. In order to set the slacking library up, we had to include the .hpp files that were downloaded from github, as well as install the libcurl library (https://curl.haxx.se/libcurl/c/libcurl.html) which is used as a C or C++ multiprotocol file transfer library. The libcurl library was very confusing to set up, and there was limited documentation available for user set up. Luckily, we were able to find a youtube video (https://www.youtube.com/watch?v=q_mXVZ6VJs4) to guide us through the process of setting up libcurl, and after following the video and some troubleshooting we were able to get it up and running. Both the slacking library and the nlohmann JSON library (https://github.com/nlohmann/json) that came packaged with it are both MIT licensed which were good to practice with in case we ever expanded on this project and wanted to create it for commercial use. After discovering Slacking, we decided to refocus our approach and figure out how we could use Slack in combination with another API. First, we thought about implementing an air quality alert system and looked at the following air quality api: https://rapidapi.com/weatherbit/api/air-quality. After examining this API we decided we wanted to go in a different direction because we wanted to retrieve a wider range of data and increase the scope of our project. We ended up choosing to use a weather API created by the same company: https://rapidapi.com/weatherbit/api/weather that allowed us to get the current weather of a location by latitude and longitude as well as other elements such as a five day forecast. Our program pulls the relevant weather measurement of temperature and provides that information to the user through a Slack message. We also monitor critical weather events such as snow, wind speed and UV ray information in order to keep our users informed and safe. In order to keep this

forecast bot constantly running, we researched ways to host our program on a virtual machine. After some research, we were able to find a virtual machine to implement with our program through the popular Microsoft Azure Cloud Computing Platform. Our program runs 24/7 on the virtual machine host allowing for remote deployment of our application.

Analysis:

The first area that we will analyze in our project is the use of the libcurl library. Libcurl "is a free and easy-to-use client side URL transfer library" that supports many different file transfer protocols. Libcurl was the library suggested by RapidAPI to use with the WeatherBit requests/response messages. We used primarily HTTP get messages implemented through the cURL library in order to request from the Weatherbit API. Below is an example of the HTTP get request we send in order to get the 5 day forecast for longitude and latitude in Spokane. As you can see, we are communicating with 52.26.32.150 using port 443 which is a port that is used for secure web browser communication. This is useful as it transfers http protocol using SSL technology (Secure Socket Layer) which operates through the exchange of security certificates making for a more secure transfer (https://www.grc.com/port_443.htm). We see many ecommerce websites these days using SSL to communicate, so we thought it was cool to use this technology in our project. As you can see in the message below, we were able to successfully connect to the API and send our GET request with our input parameters. The TCP_NODELAY set means that we will minimize the number of small packets on the network through disabling Nagle's algorithm (https://curl.haxx.se/libcurl/c/CURLOPT_TCP_NODELAY.html).

* Trying 52.26.32.150:443...

* TCP_NODELAY set

* Connected to weatherbit-v1-mashape.p.rapidapi.com (52.26.32.150) port 443 (#0)

> GET /forecast/3hourly?units=I&lang=en&lat=47.66&lon=-117.42 HTTP/1.1

Host: weatherbit-v1-mashape.p.rapidapi.com

Accept: */*

x-rapidapi-host: weatherbit-v1-mashape.p.rapidapi.com

x-rapidapi-key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

* schannel: failed to decrypt data, need more data

* schannel: failed to decrypt data, need more data

* schannel: failed to decrypt data, need more data

* Mark bundle as not supporting multiuse

Pictured below is the response message received from our API. We receive a 200 OK message back meaning that our request went through. We then are able to parse the sent content using our JSON parser inside our code and use it for our program needs.

< HTTP/1.1 200 OK

< Access-Control-Allow-Headers:

DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Cont

rol,Content-Type,Content-Range,Range

< Access-Control-Allow-Methods: GET, OPTIONS, POST

*< Access-Control-Allow-Origin: **

< Access-Control-Expose-Headers:

DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Cont

rol,Content-Type,Content-Range,Range

< Content-Type: application/json; charset=utf-8

< Date: Tue, 10 Dec 2019 07:15:06 GMT

< Server: RapidAPI-1.0.32

< X-Cache-Key:

api.weatherbit.io/v2.0/forecast/3hourly?units=I&lon=-117.42&lang=en&key=a3e3e6c71bdd4d

7586933ac1a57de143&lat=47.66

< X-Proxy-Cache: HIT

< X-RapidAPI-Region: AWS - us-west-2

< X-RapidAPI-Version: 1.0.32

< X-RateLimit-API-Requests-Limit: 150

< X-RateLimit-API-Requests-Remaining: 38

< X-RateLimit-Limit: 5100000

< X-RateLimit-Remaining: 5093324

< X-RateLimit-Reset: 1575988204

< Content-Length: 22562

< Connection: keep-alive

<

** Connection #0 to host weatherbit-v1-mashape.p.rapidapi.com left intact*

Next, we will analyze the usability and convenience of the Rapid API environment. Rapid API gives users access to thousands of API's that can be easily implemented with premade code snippets in many different languages. We have the most experience coding in C++ due to our Whitworth education, so we decided to use Rapid API's C code snippets in combination with the libcurl library to help guide our API retrieval strategies. One problem that came up while working on our project is how we were going to retrieve the response message after our get request because cURL defaults the response message just to be output in the console. After some research, we were able to find a method to write the response message into a readbuffer which then could be used with our JSON parsing library (https://gist.github.com/alghanmi/c5d7b761b2c9ab199157). Besides this minor issue, the RapidAPI code was relatively easy to implement and make the request, the JSON parsing and formatting was where we struggled the most.

For our JSON parsing, the Nlohmann library was extremely useful once we figured out exactly how to use it. One of us (Stephen) had worked with a JSON library before in a previous project, but (Griffen) had never dealt with JSON before. In order to process our JSON response message from RapidAPI, we had to use a parse function to create a JSON object from a string that was JSON formatted. We had two different types of responses: Our severe weather alert checker (which only returns one entity) and the five day forecast report (which returns 37 different entities). Because the five day forecast report returned 37 entities, in order to go to a specific time interval we had to specify which index (0) we wanted to look at. This index will give us the JSON object that contains the weather data for the next 3 hour interval our program starts. Another way we had to retrieve more specific data was used the at function with a string

parameter that directly looked at one specific attribute of the JSON object. We had to do quite a bit of back and forth conversion between JSON object and string in order to properly send out and package our get requests. Another area where we used JSON was when constructing a stylized message block for slack. We used the following documentation and guide to help us construct and test the format of these messages (https://api.slack.com/docs/message-attachments). The format of the blocks were of JSON, but we needed to insert specific data into the attributes of the JSON which provided some formatting challenges. We ended up turning the JSON into a string, and then used the find function to insert specific message data at the right spots. Then, we reconverted the string by putting it back into JSON form and attach it to the slack POST message which sends the weather report data to our specified channel in the Slack workspace.

We decided to make our own data structure that held weather attributes as well as a JSON template that would be used to make stylized message blocks. We then created a function that allowed us to merge our template with the weather data we retrieved from the API. This class made it simpler to send messages and cleaned up main as we passed in our MessageBlock object and the channel name in order to send the desired message.

We used the slacking C++ library to efficiently and easily send formatted slack messages to our Slack workspace. The slacking library works directly with the Slack WEB API Methods (https://api.slack.com/methods) in order to directly post to our workspace. We initialized a session using our specific user token, and were able to specify which channel our bot would post in. We were also able to review response messages back from Slack once we posted a message,

which was helpful for debugging purposes (especially when we accidentally posted our key to github and Slack would automatically deactivate it).

Another interesting feature we implemented in our final project was the use of a virtual machine to host our code. We used an Azure Virtual Machine imaged with Visual Studio Community 2019 in order to host our project. We were able to connect to this virtual machine through the use of RDP (Remote Desktop Protocol), a cool protocol we got to explore. RDP allows communication between Terminal Server and a Terminal server client. It is encapsulated and encrypted with TCP. For our project, we were able to use RDP and access our Azure Virtual Machine through its public IP address. Azure makes this relatively easy on us as they provide a "connect" button in their portal where we can download a pre-configured .rdp file. Once downloading this file and opening it up, the connection is requested and we are just required to enter our administrator username for the server (which was set up in the Azure Portal) as well as our password. After entering these credentials, we are able to access our virtual machine that has been pre-imaged with a copy of Visual Studio. We then download our repository off github and open it up using Visual Studio in order to run our project. When setting up the virtual machine, we went with the Dev/Test environment which is the most basic plan available. We chose a general purpose workload type. We allowed for HTTP, HTTPS, SSH, and RDP inbound ports so our virtual machine was easily accessible. For our OS disk type, we selected a standard HDD due to financial reasons, but if we had unlimited resources we would have chosen the premium SSD option to speed things up. Upon creation, we were assigned both a public and private IP address to assist us in connecting to our server.

What we learned/would improve:

Working on this final project gave us a deeper understanding of how API's work, how troublesome configuring libraries can be, hosting virtual machines, and the robust functionality of JSON. We also learned of the limited support available for C++ coupled with API programming. If we were to change our project, we would probably go back and code it in a language like Java or Python which come with more modern support for implementing with API's. We would also consider working with a slack library that is more reactive to user messages while still keeping the portability of the project in scope. We would also like to gather and report on more detailed weather information and add support for multiple locations at once.