Introduction to APIs

Lacey Conrad
MSDS 610
Regis University

June, 2020

1 Introduction

Application Programming Interfaces, or API's, are computing interfaces that seek to save time, reduce generated code, and improve consistency in the development of applications (Hoffman, 2018). Frequently, API's are used to return data, such as what we will be doing in this weeks exercise. However it is important to note that an API is not the database or server, instead it is the code that governs the access points for a server (Eising, 2017).

During application development, a plethora of tools including API's will be used to create the finished project. Quite often a developer does not need to know how an operating system builds and presents it, and this level of the tool can be abstracted. For example, think of a "save as" dialog box. A developer may need to use a "save as" dialog box in their application, but really doesn't care how the dialog box itself was created. They just need to know it is usable in their application. The "save as" dialog box is one example of an API (Hoffman, 2018).

Since a common use of an API is to request and then collect data, it isn't surprising that they're frequently used in conjunction with databases. Data can be collected, sorted, and stored in the matter most useful to the programmer. Once data is received from an API and stored in a database, it can be used in many ways, such as in a mobile application. The presence of both NoSQL and SQL databases allow for data storage options depending on the eventual use of the data.

The exercise for this week involved calling an API, receiving data, and inserting it into a database. I worked through the exercise examples first, which involved placing a call to a metaweather API. The data received from the API call was normalized into a Pandas dataframe, and data types were subjected to corrections when necessary. Then, a connection was made to a database, in this case MongoDB. The data from the dataframe was then inserted into a database that had been previously created in MongoDB Compass. To verify that the data was correctly inserted, an example document was queried from the database. I repeated a similar process with the MISO energy API, only using PostgreSQL as my database. Finally, I picked a different API to call to practice this process. I selected a Covid-19 summary API, and went through the same process including inserting the data received in the call into a MongoDB database.

2 Written portion - Questions from Week 5

1. What is an API, and what can we use them for?

   An API, or Application Programming interface, is a computing interface that defines interactions between software intermediaries. API's simplify programming by exposing only the objects a developer needs, and abstracting the underlying implementation ("Application programming interface," 2001). In many cases, they are a result of software that programmers have previously perfected, which are released to save other developers time by not having to repeat this work. This reduces code generated, and helps to create consistency across multi-application platforms (Hoffman, 2018).

   An example of this is a smart phone's photo-capture ability. If you want to capture images from the camera of a smart phone, you can use the camera's API to embed the built-in camera to your app instead of writing your own camera interface from scratch. Without being able to use an API, a programmer would have to write the camera software in addition to interpreting the input from the camera's hardware (Hoffman, 2018).

API's have four types of actions. GET requests data from a server. POST submits changes from a user to a server. PUT revises or adds information. DELETE deletes information. Through these actions, information can be searched and updated through an API (Scott, 2019).

Reasons to use APIs:

(a) Increase in productivity: API's enable workplaces to streamline their operations in addition to fostering cooperation and strengthening a company's transparency. Well-developed software components can be turned into API's and shares, enabling consistent and well-managed data exchange (Fitzgerald, 2019).

(b) User Satisfaction: API's can be used to extend the functionality of a product. In one example, a restaurant search application uses a JavaScript Maps API to provide users with an interactive map when searching for restaurants (Fitzgerald, 2019).

(c) Increase Customer/Consumer Base: When API's are shared, your network of users expands from being exclusively employees and customers to new third party developers and consumers, many of which will become reliant on the functionality provided by your API (Fitzgerald, 2019).

A few examples of APIs:

(a) Google Maps: Google Maps APIs can be used to embed Google Maps on web pages through JavaScript or Flash. The API will work on both desktop browsers and mobile browsers (Beal, 2020).

(b) YouTube: The YouTube API allows developers to embed YouTube videos in websites or applications. YouTube has several popular APIs including YouTube Analytics, YouTube Data, YouTube Livestreaming, and YouTube Player (Beal, 2020).

(c) Flickr: The Flickr photo sharing community can be accessed using the Flickr API.

(d) Twitter: Twitter has two notable APIs: the REST API which allows programmers access to twitter data, and Search API which provides developers interactions with Twitter data derived from Twitter search (Beal, 2020).

(e) Amazon Product Advertising: This API allows developers the ability to advertise Amazon products on a website in order to monetize it using Amazons product selection and discovery functions (Beal, 2020).

2. When should we consider putting API-fetched data in SQL vs a NoSQL database?

It comes as little surprise that once the data has been received from an API call, that it can be considered like any other body of data. The important questions to ask regarding should I use a SQL or NoSQL database center around how the data is going to be used. The topics we have discussed the last few weeks come into play here: how quickly does the data need to be accessed by a user, how secure does the data need to be, how structured is the data, and so on. A general framework for when SQL versus NoSQL should be used is the following:

SQL databases should be used for API-fetched data when the data fit the following characteristics: (from Joshi, 2016)

(a) Manageable data sizes

(b) Low time period or size-based retention policies

(c) Low frequency of usage

(d) Lightweight analytics

(e) Need of a standardized query interface

NoSQL databases should be used for API-fetched data when data fit the following characteristics:

(a) Scale and volume need to be flexible

(b) Deep analytics will be used

(c) There is a need for fast queries

(d) A non-standard query interface is acceptable

3. What was challenging about using APIs?

   Something I noticed after running both Jupyter examples and a few API's I called, before picking the Covid-19 API, was that they feel like you're standing in a river of data. Granted, that is what you're asking for, and the data is being received in a logical format. You initially get *all* of the data for that particular API call, and while you can pick and choose what data you insert into a dataframe (a dataframe being just one example of where the data can be saved), it felt like being inundated with columns (in particular) before sorting through what data is needed. For the "consolidated_weather" response, there are a minimum of 14 columns, and I believe there is 26 maximum columns. So far, viewing and comparing so many columns in Python/Pandas has been a little difficult for me. Also, I noticed I needed to return to the website where I found the API information to unwind abbreviations, find units, and to be able to fully decipher the API response. I don't feel like this is a great challenge though, I think I have been very impressed by the API's I have viewed and how much documentation and help each website has for allowing users to implement the API.

   However, I feel like this is complaining about being given what I asked for in the first place, and while it initially presents a challenge, I do not expect that to be the case long-term. I think the most challenging part was my inexperience with using API's, and after this exercise I feel a lot more confident on my ability to use the data from an API without being overwhelmed.

   When I was looking for an API to use for this exercise, it was somewhat challenging (as a novice) getting every API I looked to work. Most were quite simple, and the GET statement was easily executed. A few others required a key, which also wasn't too tough (especially if there were examples). However, a few API's either did not work despite my following all the literature on the website, or worked a few times and then gave an error the next day. I realize that some API's limit the number of calls you can make, but in this case, the API simply did not work from one day to the next, and I did not feel comfortable continuing to use this API if I couldn't access it again if I needed it.

3 Methods and Code

I performed this lab exercise entirely in Python (in Jupyter notebook) and utilized MongoDB as my database for the end portion of the exercise. Some output is included in the methods section where it is a logical step or verification between two parts of the exercise; dataframes will be located in the results section.

3.1 Metaweather API call and data insertion

The walk-through portion of this exercise involved calling the metaweather API and collecting and storing weather predictions for the Denver area. In order to communicate with the metaweather API, I used a GET command, and saved the resulting data in the variable response. I then viewed the data in json format, which showed the data as a key and value pair. Then I viewed the json data of the key " `consolidated_weather` " which produced a list of json objects. Each item in the list pertains to a different date.

```
import requests as req

# Communicates with the metaweather api to access current
# weather data for Denver and saves it as "response":
response = req.get('https://www.metaweather.com/api/location/2391279/')

# Displays data received during the "get" execution above:
response.json()
response.json()['consolidated_weather']
```

Table 1: An brief example of the output of `response.json()` for the raw metaweather dataset.

response.json()
{'consolidated_weather': [{'id': 4910133077344256, 'weather_state_name': 'Light Rain', 'weather_state_abbr': 'lr', 'wind_direction_compass': 'WSW', 'created': '2020-06-01T18:49:08.364987Z', 'applicable_date': '2020-06-01', 'min_temp': 19.59, 'max_temp': 31.824999999999996, 'the_temp': 30.085, 'wind_speed': 3.2504978438081604, 'wind_direction': 244.0, 'air_pressure': 1012.5, 'humidity': 24, 'visibility': 14.223807961504813, 'predictability': 75}, ...

Table 2: A brief example of the output of `response.json()['consolidated_weather]` for the metaweather dataset. This returns the data for the `consolidated_weather` key for the city of Denver.

response.json()['consolidated_weather]
[{'air_pressure': 1011.0799999999999, 'applicable_date': '2019-06-03', 'created': '2019-06-03T15:32:07.042906Z', 'humidity': 44, 'id': 6477728926662656, 'max_temp': 18.895, 'min_temp': 11.945, 'predictability': 73, 'the_temp': 18.41, 'visibility': 13.8674515827567, 'weather_state_abbr': 's', 'weather_state_name': 'Showers', 'wind_direction': 236.3759733583002, 'wind_direction_compass': 'WSW', 'wind_speed': 2.259847599045574}, ...

I then used Pandas to create a dataframe out of the data I had collected above, which gave me more general data and not day specific predictions. In order to get day specific predictions, I created an array out of the API call (using the `consolidated_weather` key), and then a dataframe for the days data. To accomplish this I created a for loop, which looped through each item in the list of json objects (which were days) and then added them to the dataframe:

```
import pandas as pd

# Creating a dataframe out of the response obtained
# from the previous code and normalizing it:
df = pd.io.json.json_normalize(response.json())
```

```
6
7  # Viewing the first 5 entries in the dataframe:
8  df.head()
9
10 # Creates an array "days:"
11 days = response.json()['consolidated_weather']
12
13 # Accesses the first entry in the "days" array:
14 days[0]
15
16 # Creates a normalized dataframe out of the first
17 # entry of the "days" array:
18 df = pd.io.json.json_normalize(days[0])
19 df.head()
20
21 # Loops through the days in the "days" array and
22 # adds each to the dataframe:
23 for day in days[1:]:
24         df = df.append(pd.io.json.json_normalize(day))
25
26 # View dataframe with all days appended on:
27 df
```

Table 3: The first entry in the days array. This array was created to collect the weather forecast for each day in order to be inserted into a dataframe.

days[0]
'id': 4910133077344256, 'weather_state_name': 'Light Rain', 'weather_state_abbr': 'lr', 'wind_direction_compass': 'WSW', 'created': '2020-06-01T18:49:08.364987Z', 'applicable_date': '2020-06-01', 'min_temp': 19.59, 'max_temp': 31.824999999999996, 'the_temp': 30.085, 'wind_speed': 3.2504978438081604, 'wind_direction': 244.0, 'air_pressure': 1012.5, 'humidity': 24, 'visibility': 14.223807961504813, 'predictability': 75

While working through this exercise, it became evident that several commands were useful to return to in order to get more information about a dataset, dataframe, or whatever aspect of data processing I was currently working with. In particular, I noticed how useful `.info()` is, especially when trying to ascertain data types, or column headers in a dataframe. Also, learning the shape of the data frame will help you to determine if your data is reflecting what you expected it to look like. For example, if you were creating a dataframe and expected 10 columns, but your shape only indicated 5, you would know something went wrong in the creation of your dataframe.

```
1  # Shows the dimensions of the data frame:
2  df.shape
3
4  # Summarizes the dataframe, showing number
5  # of entries, number of columns and their title,
6  # and data types.
```

```
df.info()
```

Table 4: The output of `df.shape`, which returns the number of rows and columns in a dataframe.

df.shape
(6, 15)

Throughout data processing, it does not appear uncommon for data types to change. For example, in our metaweather data, it is likely that the fields `created` and `applicable_date` were originally entered into a database as dates, however, over the course of being uploaded and downloaded, they became objects. Fortunately, there are ways to make sure the data types in our dataframe reflect the actual data type in the column. Also, it is not uncommon to only need certain columns out of a dataset. To save on storage space and processing power, it is a good idea to work with a data set that consists of the data you need (obviously, don't change the original data set, but have a smaller, working data set). Below, I have dropped two columns that were unneeded.

```
# Formatting the dates from objects to datetime format:
df['created'] = pd.to_datetime(df['created'], utc=True)
df['applicable_date'] = pd.to_datetime(df['applicable_date']).
        dt.tz_localize('US/Mountain')

# Dropping unwanted columns
df.drop(['weather_state_abbr', 'id'], inplace=True, axis=1)

# Making sure everything looks like I expect it to:
df.info()
df
df['created'].iloc[0]
```

Table 5: The output of df.info(), which enables us to view the columns, data types, and other pertinent information on a given dataframe.

df.info()			
<class 'pandas.core.frame.DataFrame'>			
Int64Index: 6 entries, 0 to 0			
Data columns (total 13 columns):			
weather_state_name	6	non-null	object
wind_direction_compass	6	non-null	object
created	6	non-null	datetime64[ns, UTC]
applicable_date	6	non-null	datetime64[ns, US/Mountain]
min_temp	6	non-null	float64
max_temp	6	non-null	float64
the_temp	6	non-null	float64
wind_speed	6	non-null	float64
wind_direction	6	non-null	float64
air_pressure	6	non-null	float64
humidity	6	non-null	int64
visibility	6	non-null	float64
predictability	6	non-null	int64
dtypes: datetime64[ns, US/Mountain](1), datetime64[ns, UTC](1),			
float64(7), int64(2), object(2)			
memory usage: 672.0+ bytes bytes			

At this point, I created the database in MongoDB that I will fill with data from the API. I opened my MongoDB Compass, and clicked on the create database button. I then created the `weather_test` database with the

`denver` collection, both following the naming used in the exercise. I was then able to insert my metaweather data into the `weather_test` database through the following:

```
from pymongo import MongoClient

# Connecting to the MongoDB database:
client = MongoClient()

# Defining the connection for Mongo to match to the
# databse "weather_test":
db = client['weather_test']

# Defining the connection for Mongo to match to the
# collection "denver"
collection = db['denver']

# Takes the "df" dataframe and inserts the data
# into the collection we indicated above:
df.to_dict('records')
collection.insert_many(df.to_dict('records'))

# Showing an example of a document out of the
# MongoDB collection we just inserted into:
collection.find_one()
```

3.2 Covid-19 API call and data insertion

I used an API from the COVID-19 Data Repository at `https://covid19api.com/`, which is made available by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University. I am focusing on a daily summary data set, which includes the following information about each country:

1. Total confirmed Covid-19 cases,
2. New Covid-19 deaths,
3. Total Covid-19 deaths,
4. People newly recovered from Covid-19
5. Total deaths from Covid-19
6. Total number of people recovered from Covid-19

In order to collect the API data, I followed the same procedure as above. A call was made to the API and data was collected using GET and stored in the variable response. To make sure the data was received correctly I viewed the response variable. I also viewed the data using the key `Countries`, which allowed me to see a list of json objects, which were individual countries in this case.

```
import requests as req
import pandas as pd
from pymongo import MongoClient

# Requesting a call to the covid19 API and getting
# Global summary statistics for the Covid-19 pandemic:
response = req.get('https://api.covid19api.com/summary')

# Showing the records received from the api call:
response.json()
response.json()['Countries']
```

Table 6: An brief example of the output of `response.json()` for the raw Covid-19 dataset.

response.json()
{'Global': {'NewConfirmed': 113618, 'TotalConfirmed': 6472375, 'NewDeaths': 4766, 'TotalDeaths': 387712, 'NewRecovered': 62110, 'TotalRecovered': 2757468}, 'Countries': [{'Country': 'Afghanistan', 'CountryCode': 'AF', 'Slug': 'afghanistan', 'NewConfirmed': 759, 'TotalConfirmed': 16509, 'NewDeaths': 5, 'TotalDeaths': 270, 'NewRecovered': 22, 'TotalRecovered': 1450, 'Date': '2020-06-03T20:13:50Z'}, ...

Table 7: An brief example of the output of `response.json()['Countries']` for the Covid-19 dataset which produces a list of summary statistics for each country.

response.json()
[{'Country': 'Afghanistan', 'CountryCode': 'AF', 'Slug': 'afghanistan', 'NewConfirmed': 759, 'TotalConfirmed': 16509, 'NewDeaths': 5, 'TotalDeaths': 270, 'NewRecovered': 22, 'TotalRecovered': 1450, 'Date': '2020-06-03T20:13:50Z'}, {'Country': 'Albania', 'CountryCode': 'AL', 'Slug': 'albania', 'NewConfirmed': 21, 'TotalConfirmed': 1164, 'NewDeaths': 0, 'TotalDeaths': 33, 'NewRecovered': 14, 'TotalRecovered': 891, 'Date': '2020-06-03T20:13:50Z'}, ...

I then made a dataframe out of the raw data from the API call, which showed me global pandemic statistics. Also, similar to the days looping method I used above, I created a country array, which was inserted into a dataframe. Again, I used a for loop to append each country in the `Countries` list to the dataframe.

```
# Creating a dataframe from the api data stored
```

```
# in the response variable:
df = pd.io.json.json_normalize(response.json())

# Creating a country array:
country = response.json()['Countries']

# Verifying the first item in the array:
country[0]

# Creating a dataframe for the country array:
df = pd.io.json.json_normalize(country[0])
```

Table 8: The first entry in the country array. The array will be used to loop through each country in the dataset, and append it to a dataframe created to view all country-based data.

days[0]
{'Country': 'Afghanistan', 'CountryCode': 'AF', 'Slug': 'afghanistan', 'NewConfirmed': 759, 'TotalConfirmed': 16509, 'NewDeaths': 5, 'TotalDeaths': 270, 'NewRecovered': 22, 'TotalRecovered': 1450, 'Date': '2020-06-03T20:13:50Z'}

```
# Using a for loop to loop over each item in the array
# and appending it to the "df" dataframe:
for countries in country[1:]:
        df = df.append(pd.io.json.json_normalize(country))

# Inspecting the first 50 entries of the "df" dataframe:
df.head(50)

# Viewing a summary of the "df" dataframe:
df.info()

# Formating the "Date" column:
df['Date'] = pd.to_datetime(df['Date'], utc=True)

df.info()
df.shape
```

Table 9: The output of `df.shape`.

df.shape
(34411, 10)

Table 10: The output of df.info().

df.info()			
<class 'pandas.core.frame.DataFrame'>			
Int64Index: 34411 entries, 0 to 185			
Data columns (total 10 columns):			
Country	34411	non-null	object
CountryCode	34411	non-null	object
Slug	34411	non-null	object
NewConfirmed	34411	non-null	int64
TotalConfirmed	34411	non-null	int64
NewDeaths	34411	non-null	int64
TotalDeaths	34411	non-null	int64
NewRecovered	34411	non-null	int64
TotalRecovered	34411	non-null	int64
Date	34411	non-null	datetime64[ns, UTC]
dtypes: datetime64[ns, UTC](1), int64(6), object(3)			
memory usage: 2.9+ MB			

Now that I had my Covid-19 country-specific summary data in a dataframe, I needed to insert it into MongoDB. I first went back to my MongoDB Compass and created the `covid` database and the `covid_cases` collection. I then connected to the MongoDB client, and indicated which database and collection the client should be connected to. I then used `insert_many` to insert my data into the preformed covid database. To verify the data was indeed inserted correctly, I used `find_one` to view a record:

```
# Connecting to the MongoDB client:
client = MongoClient()

# Defining the connection for Mongo to match the
# database and collection I created for the Covid-19
# dataset
db = client['covid']
collection = db['covid_cases']

# Inserting the "df" dataframe into the indicated
# collection (which is covid_cases):
collection.insert_many(df.to_dict('records'))

# Showing an example from the inserted data:
collection.find_one()
```

Now that I had my data in the MongoDB database, I was able to run a few queries on it. First, I sorted countries by number of newly confirmed Covid-19 cases in descending order:

```
# Ranking countries according to the number of newly comfirmed Covid-19 cases
# seen today (6/3/2020) in descending order:
sort = collection.find({},{"Country":1, "NewConfirmed":1, "TotalConfirmed":1})
        .sort("NewConfirmed", -1).limit(1000)

# Making a dataframe out of the results of the query:
df2 = pd.io.json.json_normalize(sort)
df2
```

Next, I queried the database to produce a list of countries who have 20000 or more deaths as a result of Covid-19:

```
# A query to return countries where there have been greater than 20000 deaths
# as a result of Covid-19:
```

```
sort3 = collection.find({'TotalDeaths':{ '$gt': 20000}}).limit(1000)

# Creating a dataframe out of the results of the query:
df4 = pd.io.json.json_normalize(sort3)
df4
```

4 Results and Output

I have included output representative of both the metaweather and Covid-19 API calls, but I did not include every dataframe and info call I ran, since many did not in fact add new information. Each dataframe/query is included as a representative of the general process I used to work from API call to database query. Many dataframes are represented with abbreviated data, since I could rarely fit it all on one vertical page.

4.1 Metaweather Data: Output

Below are the dataframes created from the raw data collected from response.jason(), and `response.json()['consolidated\_weather']`:

Table 11: A dataframe created from the raw data of the metaweather API call.

consolidated_weather	time	sun_rise	sun_set	timezone_name	...
[{'id': 4910133077344256, 'weather_state_name'...	2020-06-01T14:23:48.772657-06:00	2020-06-01T05:33:46.463364-06:00	2020-06-01T20:21:44.635222-06:00	LMT	...

Above, we see a dataframe for the raw API data, whereas below, we see the results of a dataframe created by looping through the data day by day (by looping through the list of json objects associated with the consolidated_weather key). The first dataframe displays the general response fields for the metaweather API call (time, sun_rise/set, timezone_name). Tables 12 and 13 both show the daily weather prediction data by calling the consolidated_weather key, specific to Denver.

Table 12: The creation and first entry of a Pandas dataframe which will hold daily metaweather data. In this case, it shows the predicted weather for 6/1/2020 in Denver, CO.

weather_state _name	weather_state _abbr	wind_direction _compass	created	applicable_date	min_temp	max_temp
Light Rain	lr	WSW	2020-06-01T 18:49:08.364987Z	2020-06-01	19.59	31.825

Table 13: A dataframe of metaweather data received from API call on 6/2/2020, showing the weather forecast for 6/1/2020 through 6/6/2020 for Denver, CO.

weather_state _name	weather_state _abbr	wind_direction _compass	created	applicable_date	min_temp	max_temp
Light Rain	lr	WSW	2020-06-01T	2020-06-01	19.590	31.825
Heavy Cloud	hc	NNW	2020-06-01T	2020-06-02	18.990	31.130
Showers	s	SSE	2020-06-01T	2020-06-03	18.525	31.295
Light Cloud	lc	SSW	2020-06-01T	2020-06-04	18.660	32.170
Heavy Cloud	hc	SE	2020-06-01T	2020-06-05	17.850	33.705
Clear	c	S	2020-06-01T	2020-06-06	18.100	32.890

The following is a document returned from the weather_test database within MongoDB using the `find_one` command. This document is the data pertaining to the weather forecast on 6/1/2020:

Table 14: A query returning an example of a document within the MongoDB metaweather database. In this case, the document is the weather forecast for 6/1/2020.

collection.find_one()
{'_id': ObjectId('5ed56641b0caedb9ebe17db3'), 'weather_state_name': 'Light Rain', 'wind_direction_compass': 'WSW', 'created': datetime.datetime(2020, 6, 1, 18, 49, 8, 364000), 'applicable_date': datetime.datetime(2020, 6, 1, 6, 0), 'min_temp': 19.59, 'max_temp': 31.824999999999996, 'the_temp': 30.085, 'wind_speed': 3.2504978438081604, 'wind_direction': 244.0, 'air_pressure': 1012.5, 'humidity': 24, 'visibility': 14.223807961504813, 'predictability': 75}

4.2 Covid-19 Dataset: Output

Similar to the metaweather dataset, I created a dataframe out of the raw data from the Covid-19 API call:

Table 15: A normalized dataframe created from the raw data of the Covid-19 API call.

Countries	Date	Global.New .Confirmed	Global.Total .Confirmed	Global.New .Deaths	Global.Total .Deaths	Global .NewRecovered
[{'Country': 'Afghanistan', 'CountryCode': 'AF...	2020-06-03T20:13:50Z	113618	6472375	4766	387712	62110

As can be ascertained from the titles, this dataframe shows the global Covid-19 summary statistics. In particular, I note 6,273,475 global confirmed cases of Covid-19, and 387,712 deaths due to the virus. In order to look at the summary statistics of each country, I created another dataframe using the Countries sub-response field.

Table 16: The creation and first entry of a dataframe which will have country-specific Covid-19 data inserted into it.

Country	Country Code	Slug	NewConfirmed	TotalConfirmed	NewDeaths	TotalDeaths	
Afghanistan	AF	afghanistan	759	16509	5	270	...

The Covid-19 cases per country dataframe was the first place I started noticing some oddities with my data. While it is not shown in the following dataframe, I saw in some cases several countries being listed multiple times in the dataframe with identical data. It did not happen predictably, some countries had a great deal more copies than others (the United States, for example, seemed to have 500 entries), and many countries had one entry. I tried to look up methods to consolidate these entries (since clearly they were repetitions), yet could not get any to work.

Table 17: A dataframe of Covid-19 data (total cases, new cases, etc) per country, received from API call on 6/3/2020.

Country	CountryCode	Slug	NewConfirmed	TotalConfirmed	NewDeaths	TotalDeaths	...
Afghanistan	AF	afghanistan	759	16509	5	270	...
Albania	AL	albania	21	1164	0	33	...
Algeria	DZ	algeria	113	9626	6	667	...
Andorra	AD	andorra	79	844	0	51	...
Angola	AO	angola	0	86	0	4	...
...	...	...	...	...	...	...	...

For the sake of curiosity, I did a little math to see the population sizes of the countries listed below to determine what percentage of their population has been sick. As we can see, the Southern European country of Andorra has the largest percentage sick based on their population, whereas the Southwestern African country of Angola has the least number of sick:

Table 18: A dataframe of Covid-19 data (total cases, new cases, etc) per country, received from API call on 6/3/2020.

Country	Population	TotalConfirmed	Percent Population (Total Confirmed)	TotalDeaths
Afghanistan	37,170,000	16509	0.04%	270
Albania	2,846,000	1164	0.04%	33
Algeria	42,230,000	9626	0.02%	667
Andorra	77,006	844	1.09%	51
Angola	30,810,000	86	2.79e-4%	4
...	...	...	...	...

The results of the `find_one()` command used on the covid database in MongoDB via my pymongo connection are the following:

Table 19: The output of `collection.find_one()` for data within the MongoDB Covid-19 database.

collection.find_one()
{'_id': ObjectId('5ed80087a85804cb91f9aa0b'), 'Country': 'Afghanistan', 'CountryCode': 'AF', 'Slug': 'afghanistan', 'NewConfirmed': 759, 'TotalConfirmed': 16509, 'NewDeaths': 5, 'TotalDeaths': 270, 'NewRecovered': 22, 'TotalRecovered': 1450, 'Date': datetime.datetime(2020, 6, 3, 19, 12, 45)}

The first of the two queries I ran sorted countries by newly confirmed cases, in descending order. This output shows the multiple entries some countries had in the dataset, which prevented me from getting a clean sort of each countries rank in the number of new Covid-19 cases, although it is clear that Brazil had the most new cases on 6/3/2020.

Table 20: The output of a MongoDB query on the Covid-19 dataset, sorting the data by amount of newly confirmed cases, in descending order.

	_id	Country	NewConfirmed	TotalConfirmed
0	5ed80087a85804cb91f9aadd	Brazil	28936	555383
1	5ed80087a85804cb91f9ab97	Brazil	28936	555383
2	5ed80087a85804cb91f9ac51	Brazil	28936	555383
3	5ed80087a85804cb91f9ad0b	Brazil	28936	555383
4	5ed80087a85804cb91f9adc5	Brazil	28936	555383
...	...	...	...	...
994	5ed808b9eab414c65a3c7ff1	United States of America	20461	1831821
995	5ed808b9eab414c65a3c80ab	United States of America	20461	1831821
996	5ed808b9eab414c65a3c8165	United States of America	20461	1831821
997	5ed808b9eab414c65a3c821f	United States of America	20461	1831821
998	5ed808b9eab414c65a3c82d9	United States of America	20461	1831821
...	...	...	...	...

Here is the query returning countries who have had greater than 20000 deaths due to Covid-19. It was my hopes to sort this query as well, however, that produced the same repeated country entries. Without sorting, I was able to manually weed out some of the duplicates, and came up with an initial list (although it is not inclusive and not sorted).

Table 21: The output of a MongoDB query that returns all countries which have had greater than 20000 deaths due to Covid-19.

Country	CountryCode	Slug	NewConfirmed	TotalConfirmed	NewDeaths	TotalDeaths	
France	FR	france	0	189348	107	28943	...
Italy	IT	italy	318	233515	55	33530	...
Spain	ES	spain	294	239932	0	27127	...
United Kingdom	GB	united-kingdom	1656	279392	325	39452	...
United States of America	US	united-states	20461	1831821	1015	106180	...
Brazil	BR	brazil	28936	555383	1262	31199	...
...	...	...	...	...	...	...	...

5 Analysis

5.1 Data Analysis

The exercise for this week mostly pertained to obtaining and storing data, and as such, had little in the way of data analysis. This does not mean the data did not provide valuable information. In the case of the metaweather data, we can see the changing weather conditions day by day for the next several days. Data of this form is valuble as it is, although if needed, analysis could be performed to should percentage increase in temperature, relative humidity, wind chill, etc.

The Covid-19 data was very interesting to look at, and I really wished I wasn't having the problem with duplicated entries, because I would have liked to have the ability to work with the entire data set. The conclusions we can gleam from what data I have displayed are that Europe and North America have been particularly hard hit from Covid-19, although it is possible with more data that we would see other hot-spots. Also, South America seems to be currently producing most of the worlds new cases of Covid-19, however, the United States is not far behind them.

5.2 Analysis of API's

The usage of the API's was generally straight forward, and the documentation on websites the API's derived from was very helpful and descriptive of the fields of data that would be present. It did feel like a big dump of data, but that is the intention, so I cannot fault the API creators for that. The data was also fairly well structured, and when organized in a dataframe, was significantly less overwhelming. It was also easy to format the data so that only the data needed for analysis or insertion into a database were saved in the dataframe.

6 Conclusion

API's are a very valuable tool for any programmer. They can save time, and help improve consistency in various aspects of the development of an application. And since they can be interacted with through one line of code, they seem to be rather easy to use. It is also very helpful that many API websites give you more refined options of where you can GET their data, which likely keeps some unnecessary data from being downloaded.

I can see the need to observe the process of incorporating a new API into a project. The chances that all the data (or other medium) will be needed in a new application are small, and the process of saving a large data set to a local PC, and then uploading it as-is to a database can take up valuable storage space and processing power. Being able to use something like a dataframe as an intermediary to prevent bogging down a database seems like a useful tool, and I imagine once I become more versed in API's that it will be easier to manage what is initially obtained with GET.

I believe one of the more powerful applications of a technology like this is the ability to receive data (such as we did) very rapidly in a big chunk. It is still overwhelming to receive so much data, yet, in cases where time is of the essence, such as processing Covid-19 data, a user can go from GET statement to Database insertion in a matter of minutes. And from there, queries can be ran. Unless I'm mistaken, you could perform your API call at 9am, and by lunchtime, have some basic statistics ready to be used however. This quick turn-around, and API's ease of use, really make this a great adaption to data analysis.

7 References

1. *Application programming interface.* (2001, July 30). Wikipedia, the free encyclopedia. Retrieved June 4, 2020, from https://en.wikipedia.org/wiki/Application_programming_interface

2. Beal, V. (2020). *What is API - Application program interface? Webopedia definition.* Webopedia. https://www.webopedia.com/TERM/A/API.html

3. Eising, P. (2017, December 7). *What exactly IS an API?* Medium. https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f

4. Fitzgerald, A. (2019, October 27). *The ultimate guide to accessing & using APIs.* HubSpot Blog. https://blog.hubspot.com/website/application-programming-interface-api

5. Hoffman, C. (2018, March 21). *What is an API?* How-To Geek. https://www.howtogeek.com/343877/what-is-an-api/

6. Joshi, V. (2016, August 4). *Relational vs. NoSQL databases for API traffic.* dzone.com. https://dzone.com/articles/relational-vs-nosql-databases-for-api-traffic-1

7. Scott, T. (2019, January 11). *How to use an API: Just the basics.* TechnologyAdvice. https://technologyadvice.com/blog/information-technology/how-to-use-an-api/