Title: **Lab 01 Report**
Notice: Dr. Bryan Runck
Author: **Michael Felzan**
Date: **February 11, 2021**

Project Repository: https://github.com/fezfelzan/GIS5572.git

Abstract

This report demonstrates how Python code written in Esri Jupyter Notebooks using the Python "requests" library may interact with various APIs and extract data to a local device. A methodology for extracting data from the Minnesota Geospatial Commons' CKAN API is first explained, which utilizes HTML POST requests. Then, the process on how to programmatically receive data from the Google Places API—an API which requires a key—is outlined. Finally, this document demonstrates how to extract North Dakota weather data from the NDAWN API. An evaluation of the effectiveness of each process is provided at the end of this document.

Problem Statement

The objective of this lab was to build a set of ETL pipelines which programmatically extract data from the Minnesota Geospatial Commons API, the Google Places API, and the NDAWN API. An overview of the APIs was to be provided in this document, along with models and descriptions which deconstruct the innerworkings of the three interfaces. The HTML data extraction processes used were to be performed in an Esri Jupyter Notebook environment, and then presented and explained in this document.

Figure 1. Infographic depicting the lab objective: programmatically extracting data from APIs

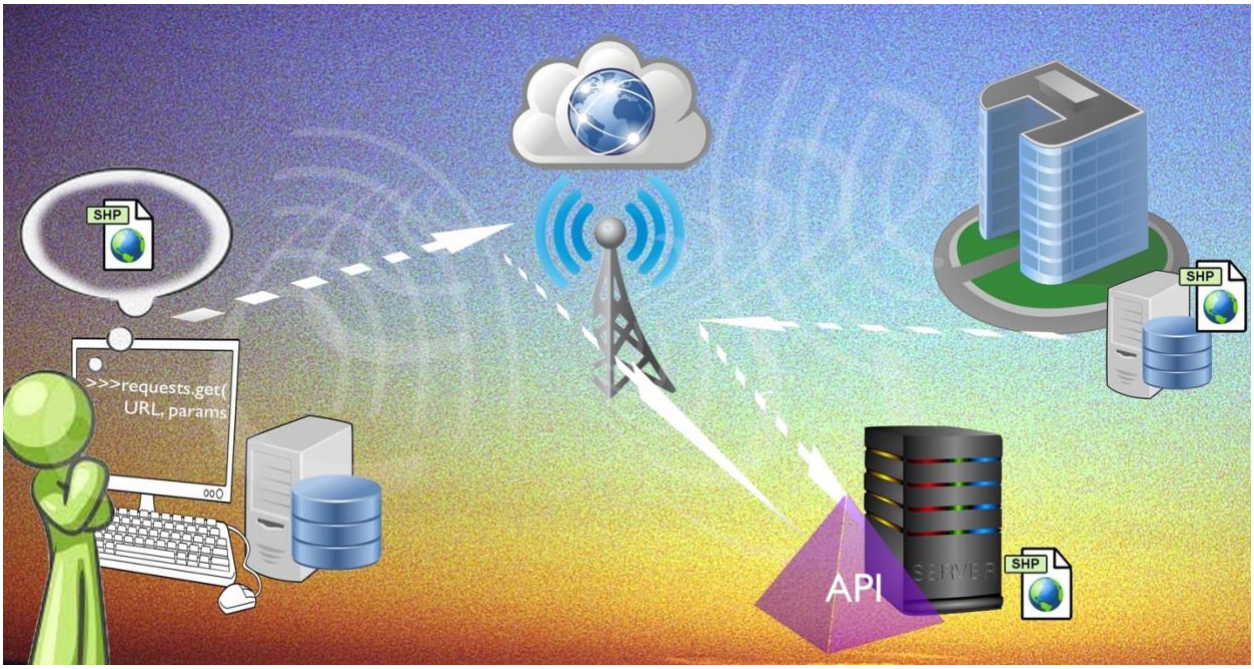


Table 1. ***Elements required to complete the Lab 1 objective.***

#	Requirement	Defined As	Spatial Data	Attribute Data	Dataset	Preparation
1	Coding Environment	Esri Jupyter Notebooks				Making sure you have access to the ArcGIS3 environment
2	Google Cloud Account Membership	Acess to Google Places API Key				Signing up for Google Cloud
3	Wifi Access					Finishing moving into your new apartment

Input Data

There was no input data required for this Lab – just access to Esri Jupyter Notebooks, the requests library, and the web.

I. Minnesota Geospatial Commons and the CKAN API

A. Overview of the API

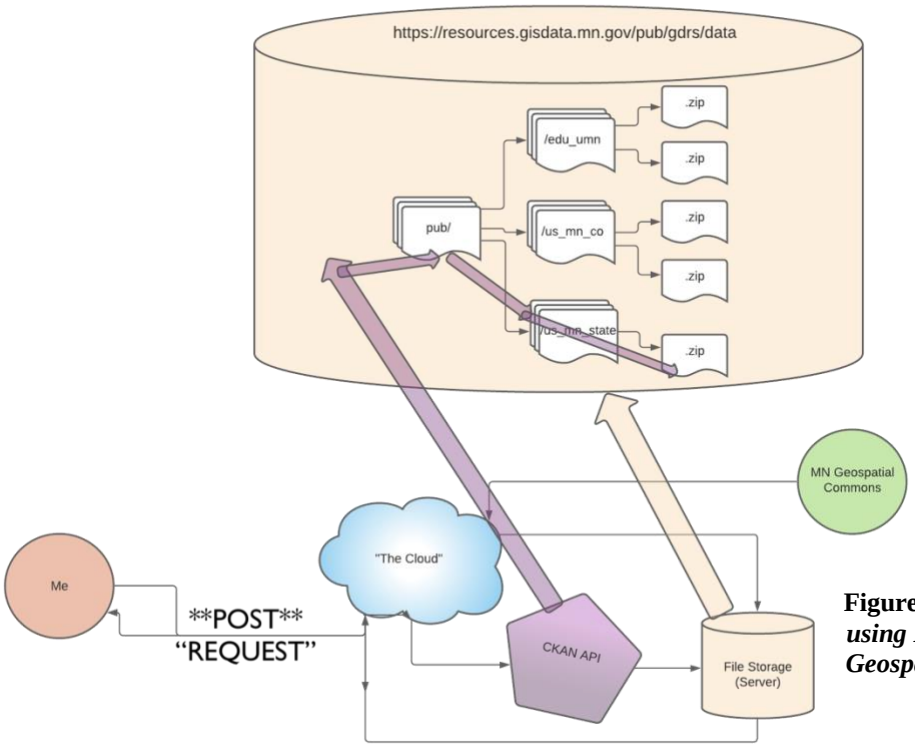


Figure 2. ***Flow map depicting the process of using POST requests to extract data from the MN Geospatial Commons CKAN API.***

An application programming interface ("API") is the liaison between you (the user) and the server you wish to interact with (get data from, post data to, etc.). In a Jupyter Notebook environment, on a device that has internet connectivity, you may send an 'abstract' (binary) message over "the web" and have it be received by the API associated with the server you wish to extract data from. There are many ways an API can be set up, but each has its own ruleset, or its own internal structure and way of processing commands and actions, that you, the user, must adhere to when sending it a message.

The Minnesota Geospatial Commons runs on the CKAN API, which has a built-in "REST-ful" (Representational State Transfer) API (https://gisdata.mn.gov/content/?q=help/api), which means it uses HTTP requests to interact with the data on the MN Geospatial Commons **(5).**

An important facet to the CKAN API is that it uses POST requests, instead of GET requests, to access the data (along with the functionality of being able to pass a dictionary of parameters) **(3).** This methodology differs from the two other APIs we will discuss in the following sections.

The Minnesota Geospatial Commons website is set up in a hierarchical way, where:

gisdata.mn.gov/

is the home page, and adding "content/" onto the end of the URL base will take you to a different "directory:"

gisdata.mn.gov/**content/**

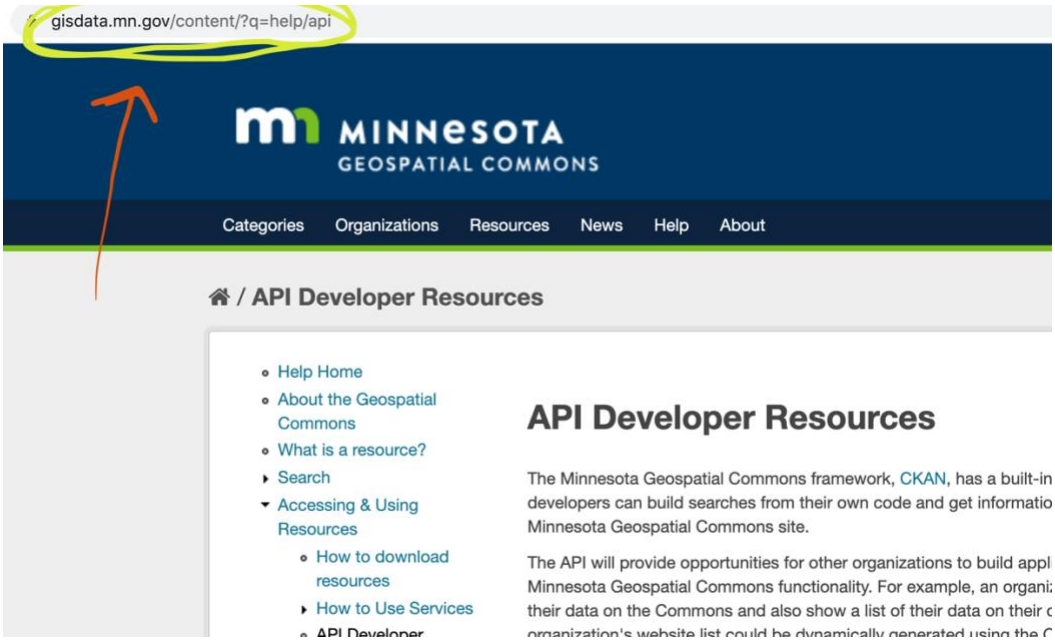


Figure 3. ***MN Geospatial Commons webpage and corresponding URL path***

This, I believe, is much like a directory tree on your home computer's file explorer. If we think of "gisdata.mn.gov" as the base folder, "gisdata.mn.gov/content/" is going a level deeper… however, much like on a directory tree, you may have options of what folder to choose once you enter another folder.

Keeping this in mind, we will navigate the MN Geospatial Commons website and access data using the website's GUI. When we click "Resources," and then search and click on the layer we wish to download, the GUI navigates us to this page/URL:

https://gisdata.mn.gov/dataset/bdry-planning

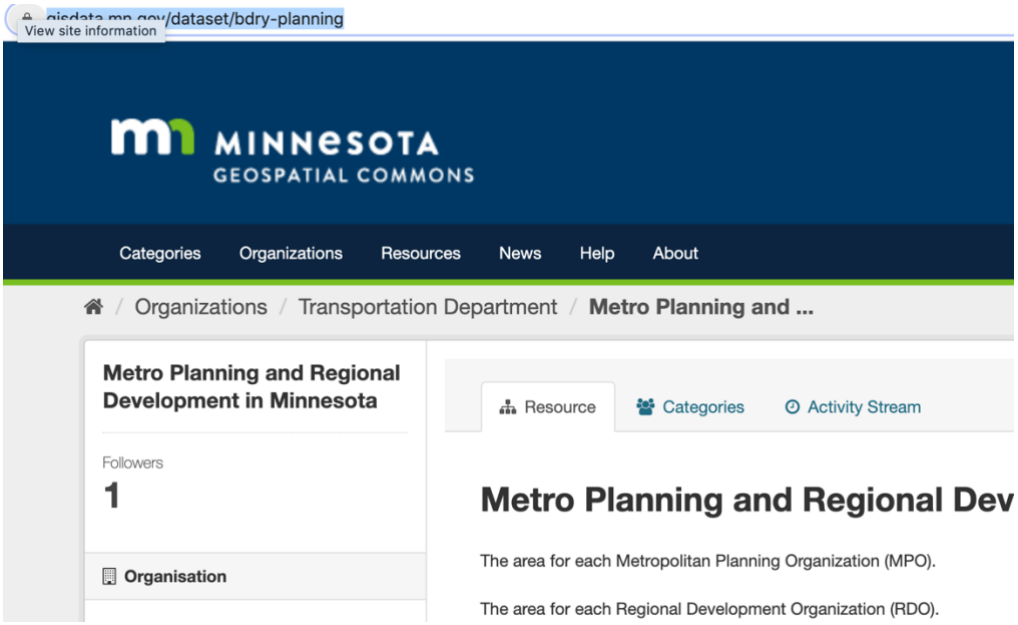


Figure 4. ***demonstrating how the components of the MN Geospatial Commons URL describe a hierarchical organization***

We can see here that we have moved directories – we are no longer in the "content/" directory, and we are now in **"dataset/"**

However, if we sent a request to the API for this page, we would not get the actual data we're interested in – we would only receive the text on this page. What we're *really* interested in is what the link address to the Shapefile looks like, and we can get that by right-clicking "Copy Link Address" on the data download button/link :

Figure 5. ***Copying the link address to a shapefile's download link to view its URL structure***

Copy-pasting that link into a text file, we receive this:

https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dot/bdry_planning/shp_bdry_planning.zip

This is interesting, because it is not the same base as the MN Geospatial Commons webpage – the base of the URL is "resources.gisdata.mn.gov/" instead of "gisdata.mn.gov/." So what happens when we punch in *just* the base of this URL to a search engine?

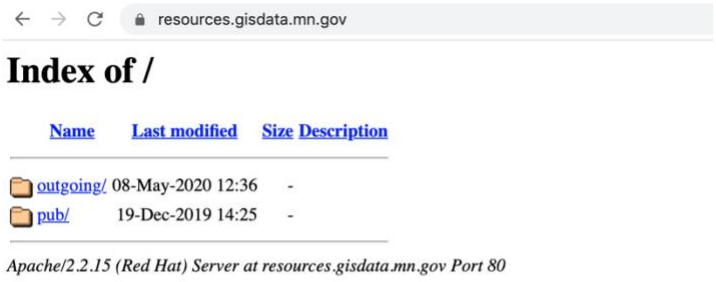


Figure 5. ***entering in "resources.gisdata.mn.gov" into a search browser reveals the file "tree" structure of the MN Geospatial Commons database***

This brings us to, quite literally, a folder directory tree, built off of URL extensions (entering the name of the folder at the end of the URL you're currently on brings you inside that folder). Further navigating this structure, we may find ourselves with the opportunity to download a .zip file of some Minnesotan data:

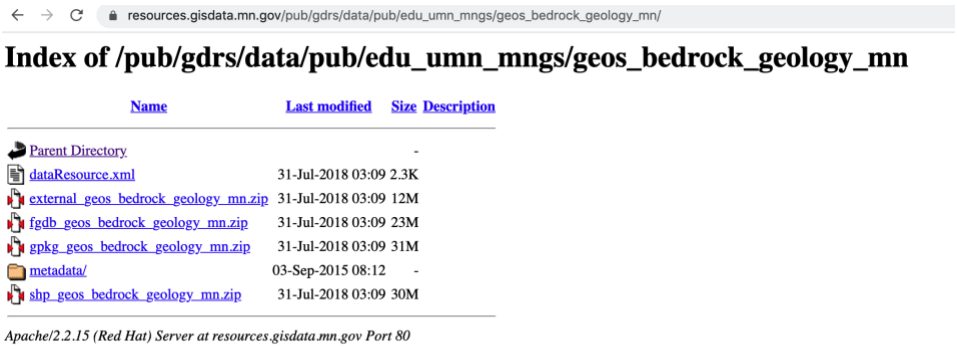


Figure 6. example of an end to a file tree "branch," where shapefiles reside

So, knowing that we can navigate to any data file if we know the directory path, all we need to do is programmatically access all file options in a given folder (the equivalent of saying "list files" at the command line) print off all possibilities, and we can navigate to any file we choose via string concatenation in Jupyter Notebooks. However, because we are not interfacing with this data in a true directory folder structure, but rather via HTML pages, we need to scrape the data on these pages to receive the names of each item.

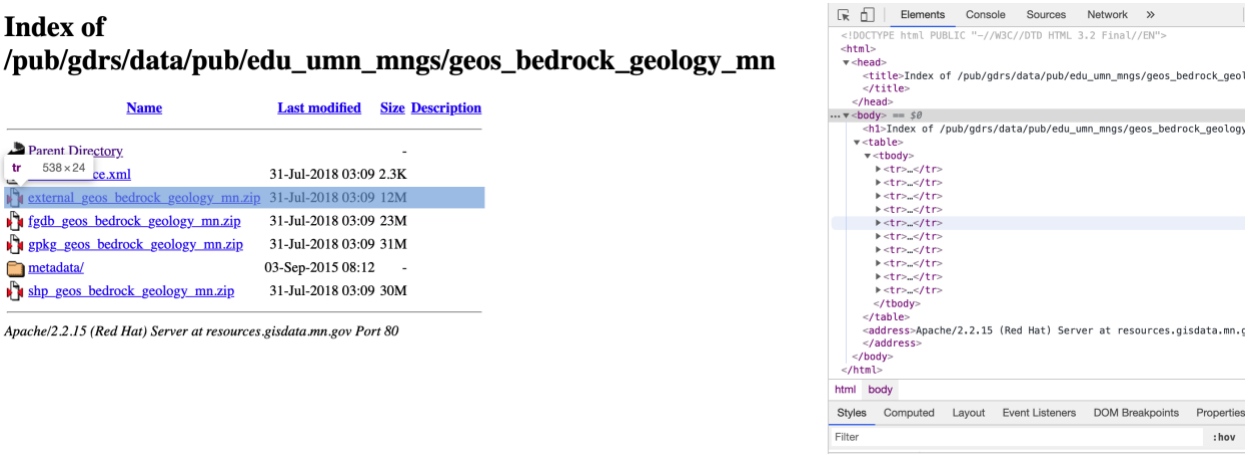


Figure 7. Using Google Chrome's Developer Tools to identify which part of the HTML text holds the name for each data item

If we sent a request for the HTML page we're currently on, we'd receive all of the text of what's on the page…of course, embedded in a slew of other ascii. This is where an HTML parser (like Beautiful Soup 4) *reaaally* would come in handy, but, because we are working in a Jupyter Notebook running the ArcGIS environment, we cannot import or install BS4.

Instead, we can try to use Google Chrome's Developer Tools to parse out which specific part of the HTML (and all related pages in this structure) relate to the text items we're interested on the page, and programmatically print them out cleanly. The following section will explain my process in doing so.

B. Method of interfacing with the CKAN API

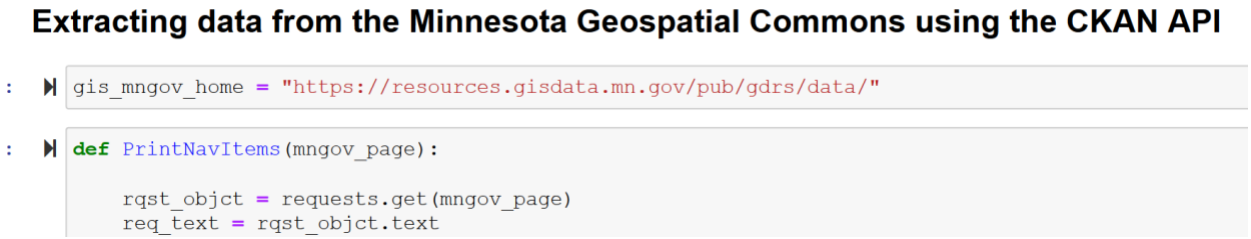


Figure 7. running a request.get() on a page to extract the whole page's HTML text

Our first step is to run a basic get request on a page in the MN Geospatial Commons file structure. **Figure 7.** demonstrates how we would do this: use the *requests* library to run a GET on a URL string, and save the returned request object to a variable. Then, we can use the ".text" function in the requests library to turn our request object into text.

If we printed this new variable, we would see a massive sea of barely legible characters and words. For this particular page, the text is broken up by newlines (\n). To start the process of

parsing, I decided to break the code up by newlines (returning the variable as a list of broken-up components).

```
split_by_newlines = req_text.split("\n")
```

The next part may be a little painful to explain, because the process is completely specific to these series of HTML pages. But… I then iterated over every item in the list of HTML chunks, and passed some statements saying, "if the text chunk has this pattern, append the chunk to a new list. If not, do nothing."

```
def PrintNavItems(mngov_page):

    rqst_objct = requests.get(mngov_page)
    req_text = rqst_objct.text
    split_by_newlines = req_text.split("\n")

    relevent_items = []

    for item in split_by_newlines:
        if "?C=N;O=D" in item:
            pass
        elif "/pub/gdrs/" in item:
            pass
        elif "a href=" in item:
            relevent_items.append(item)

    for item in relevent_items:
        firstsplit = item.split('<a href="')[1]
        sec_split = firstsplit.split('">')[0]
        print(sec_split)
```

Figure 8. function which parses a MN Geospatial Commons page's HTML text for the "cleaned up" names of the data items

This function, using a series of string splitting steps, extracts all of the "a href=" data items on the page, excluding the header/descriptor items, and prints all relevant page items out in a new cell. Having created this function, we can now just use string concatenation to navigate folders, and then use the function to print out all of the possibilities of where to move next in that directory:

```
PrintNavItems(gis_mngov_home)
```

```
pub/
```

```
user_selection = "pub/"
```

```
new_concatenation = gis_mngov_home + user_selection
PrintNavItems(new_concatenation)
```

```
com_mvta/
edu_umn/
edu_umn_mngs/
edu_umn_nrri/
org_mmcd/
org_mn_brrwd/
org_mn_lqpybwd/
org_mn_mesb/
org_mn_nfcrwd/
org_mn_wrwd/
us_mn_co_blueearth/
us_mn_co_brown/
us_mn_co_chippewa/
us_mn_co_dakota/
us_mn_co_faribault/
us_mn_co_freeborn/
us_mn_co_itasca/
us_mn_co_lake/
us_mn_co_lesueur/
us_mn_co_mcleod/
us_mn_co_meeker/
us_mn_co_ramsey/
```

Figure 9. Running the function created in Fig. 8

Once we've found the path to the file we're looking for, we can run a **POST** request on the URL path, and save the response to memory (a variable). Having saved the request object to a variable, we can then encode the data properly (using io.BytesIO), and extract the zip file to a path on our own disc storage (with some help from the ZipFile package):

```
user_selection = "us_mn_co_itasca/"
```

```
new_concatenation = new_concatenation + user_selection
PrintNavItems(new_concatenation)
```

```
plan_parcels.zip
plan_parcels/
plan_sections.zip
plan_sections/
plan_townships.zip
plan_townships/
plan_zoning.zip
plan_zoning/
```

```
zip_selection = "plan_sections.zip"
```

```
final_path = new_concatenation + zip_selection
final_path
```

```
]: 'https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_co_itasca/plan_sections.zip'
```

```
output_path = r"C:\Users\michaelfelzan\Documents\5572_Lab01"
```

```
post_req_obj = requests.post(final_path)
zippy = zipfile.ZipFile(io.BytesIO(post_req_obj.content))
zippy.extractall(output_path)
```

II. Google Places API

A. Overview of the API

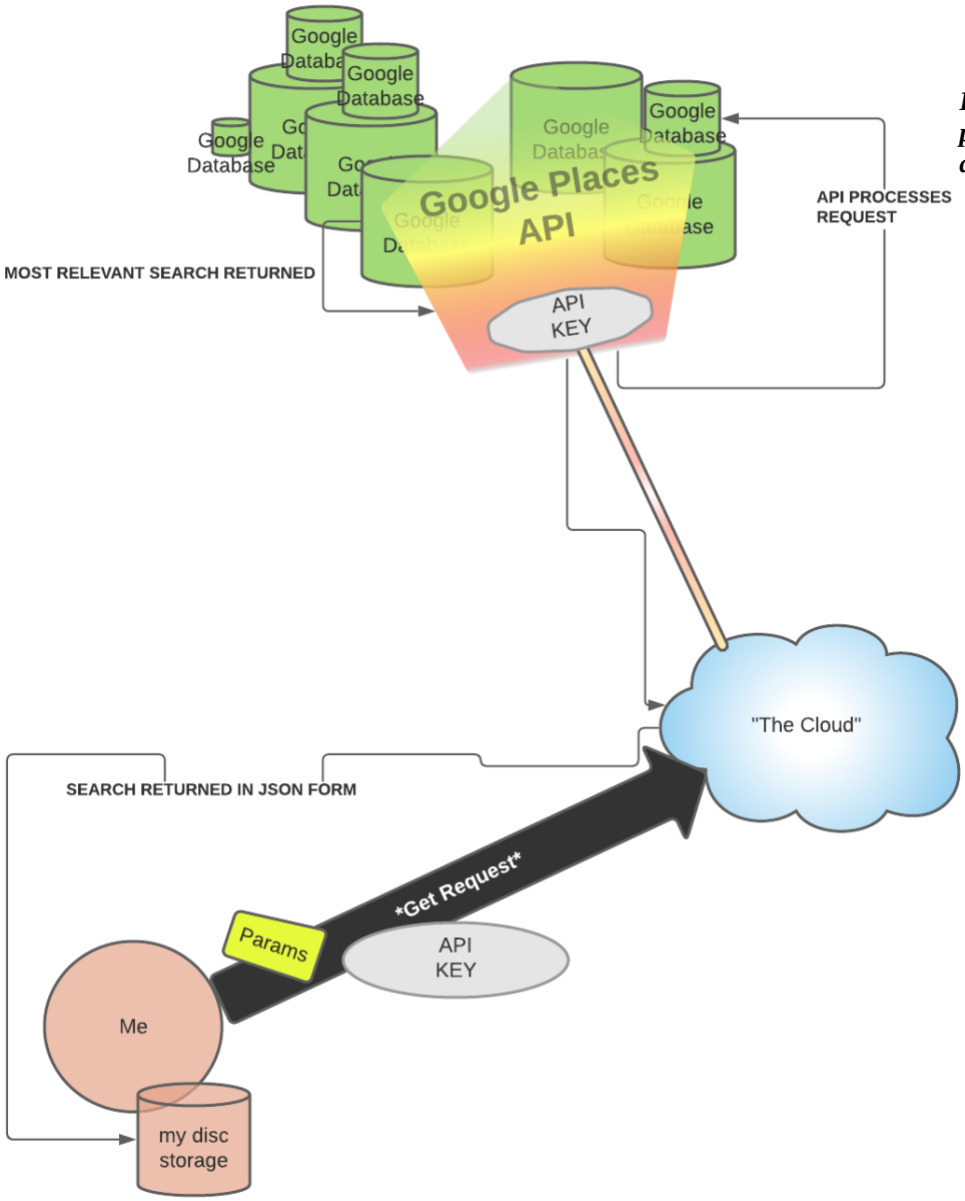


Figure 10. Flow map depicting the process of using GET requests and an API key to extract data from the MN Geospatial Commons CKAN API.

The Google Places API is similar to the CKAN API, in that it uses HTTP requests to return information (2). However, this API differs from the two other APIs on this list in that you need your own API ***key*** to receive information back from the interface. To generate your own key, you can create a Google Maps Platform account and sign up for the free trial (but still have to enter in your credit card information, I believe). The steps for generating your own key have been outlined by Google here: (1)

The Google Places API is designed so that you may send a place name (eg. “Mall of America”) in a GET request, along with whatever attributes you want about that place (Details, Photos, etc.), and you assign those attributes as parameters in the request you make.

B. Method of interfacing with the Google Places API

Extracting data from the Google Places API

```
my_API_key = "AIzaSyDFqX3I0LQmRZ23WJwBnnh7Y5VZZpwYt7o"
```

```
goog_places_URL_base = "https://maps.googleapis.com/maps/api/place/findplacefromtext/json"
```

```
user_input_place = "Mall of America"
```

```
formatted_input = user_input_place.replace(" ", "%20")
formatted_input
```

```
5]: 'Mall%20of%20America'
```

```
# P A R A M S

input_param = "?input=" + formatted_input
input_type = "&inputtype=textquery"
fields_param = "&fields=photos,formatted_address,name,rating,opening_hours,geometry"
key_param = "&key=" + my_API_key
```

```
final_goog_path = goog_places_URL_base+input_param+input_type+fields_param+key_param
final_goog_path
```

```
7]: 'https://maps.googleapis.com/maps/api/place/findplacefromtext/json?input=Mall%20of%20America&inputtype=textquery&fi
elds=photos,formatted_address,name,rating,opening_hours,geometry&key=AIzaSyDFqX3I0LQmRZ23WJwBnnh7Y5VZZpwYt7o'
```

Fig 11. Setting up URL to pass into a GET request to the Google Places API

Figure 11. outlines the process of how I constructed my GET request for the Google Places API. The first step is assigning your API key to a variable. Next, I assigned the URL base (format?) for the requests to a variable, to later concatenate it to another string. For my input place, I assigned a variable to the string, “Mall of America,” and then formatted the string to have “%20” in place of where the whitespaces are, which I believe is necessary formatting to use this API. Next, I set my formatted input equal to the input= variable, and added the string key/value pairs for the other variables at the end of the URL string as well. Then, I sent the final URL as a GET request, and received back the information in req. object form, which I finally converted into text:

```
In [98]:  goog_req_obj = requests.get(final_goog_path)

          print(goog_req_obj.text)
```


```
{
   "candidates" : [
      {
         "formatted_address" : "60 E Broadway, Bloomington, MN 55425, United States",
         "geometry" : {
            "location" : {
               "lat" : 44.8548651,
               "lng" : -93.2422148
            },
            "viewport" : {
               "northeast" : {
                  "lat" : 44.85850560000001,
                  "lng" : -93.23805889999998
               },
               "southwest" : {
                  "lat" : 44.85066959999999,
                  "lng" : -93.24699650000002
               }
            }
         },
         "name" : "Mall of America®",
         "opening_hours" : {
            "open_now" : false
         },
         "photos" : [
            {
               "height" : 6936,
               "html_attributions" : [
                  "\u003ca href=\"https://maps.google.com/maps/contrib/105728221807217974918\"\u003eWin Winst\u003c
/a\u003e"
               ],
               "photo_reference" : "ATtYBwJIF4_j1vTFk0YiP4Rh--LMj9zONeOaAJ7s-Eqr1E9Y2YNb5r85i1kexjUTywYBuaVMGqaSxQO
17iTu1OdqdevvllLl33GE1qZoxEfBVYSuI5ruwuYKcAFvogBFsmiyNb-AJXma0URp2fQ-DksmyuY0qubjgDVFJlCgRggZa9XqS77B",
               "width" : 9248
```


Fig 12. Info sent back about the "Mall of America" from the Google Places API

III. NDAWN API

A. Overview of the API

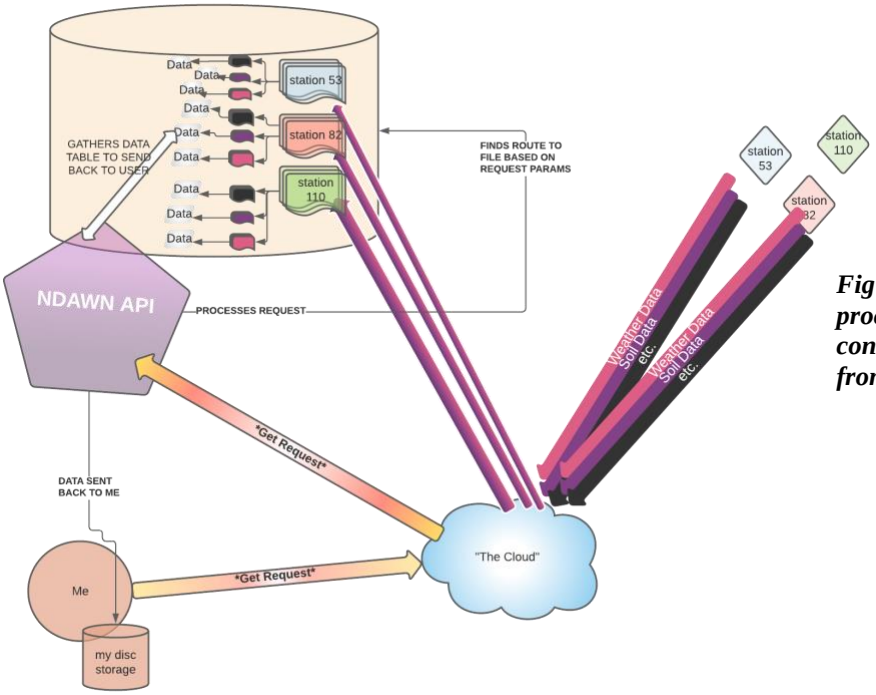


Fig 13. Flow map depicting the process of using GET requests containing URL params to extract data from the NDAWN API

Unlike the Google Places API, the NDAWN (North Dakota Agricultural Weather Network) API does not require an API key. This API also uses HTTPs requests to process and gather information.

NDAWN gathers data in near- real-time from network of weather stations around North Dakota. Each station sends its own daily reports on air temperature, soil data, wind speed, dew point, etc., and the data is stored in a tabular form on a server.

To begin the process of data extraction, we may follow the theory/ methodology outlined in Section I. to copy the link address of a specific piece of data's download link:

https://ndawn.ndsu.nodak.edu/table.csv?station=10&variable=mdws&year=2021&ttype=monthly&quick_pick=&begin_date=2020-02&count=12

Here, we can see the base of the URL (everything up to the "?") is

https://ndawn.ndsu.nodak.edu/get-table.html?
and the variable immediately after this URL base is the specific weather station we chose to get data from. It may be safe to assume that the NDAWN database is organized in a hierarchical structure, where the base of the directory, there are "folders" for each weather station, and the various types of data for each day are subcomponents to each folder. Because the link to this data is just a series of URL parameters **(6),** where mdws = the type of weather data you're requesting, ttype = the time interval, etc., we can just pass a dictionary of key:value pairs that correspond to the exact names of the URL parameters, in no specific order, as the second argument in our GET request function, and we should receive back a .csv of the specific station/data type/timeframe we requested.

B. Method of interfacing with the API

In [50]:
```
master_params = {
    "station" : "110",
    "variable" : "mdavt",
    "dfn" : "",
    "year" : "2021",
    "ttype" : "monthly",
    "quick_pick" : "",
    "begin_date" : "2020-02",
    "count" : "12"}
```

In [51]:
```
base_ndawn_url = "https://ndawn.ndsu.nodak.edu/table.csv"
```

In [52]:
```
reqreq = requests.get(base_ndawn_url, params = master_params)
```

In [53]:
```
with open("Testtesttest.csv", "w") as test_txt:
    test_txt.write(reqreq.content.decode('utf-8'))
```

In [58]:
```
peek = pd.read_csv('Testtesttest.csv', skiprows = 5)
peek.head()
```

Out[58]:

	Station Name	Latitude	Longitude	Elevation	Year	Month	Avg Temp	Number Missing	Number Estimated	Monthly Normal Average Air Temperature	Departure from Normal Monthly Average Air Temperature	Number Missing.1	Number Estimated.1
0	NaN	deg	deg	ft	NaN	NaN	Degrees F	NaN	NaN	Degrees F	Degrees F	NaN	NaN
1	Zeeland	46.013378	-99.687587	2070	2020.0	2.0	15.701	0.0	0.0	17.00	-1.299	0.0	0.0
2	Zeeland	46.013378	-99.687587	2070	2020.0	3.0	29.808	0.0	0.0	28.00	1.808	0.0	0.0
3	Zeeland	46.013378	-99.687587	2070	2020.0	4.0	37.843	0.0	0.0	43.00	-5.157	0.0	0.0
4	Zeeland	46.013378	-99.687587	2070	2020.0	5.0	52.073	0.0	0.0	55.00	-2.927	0.0	0.0

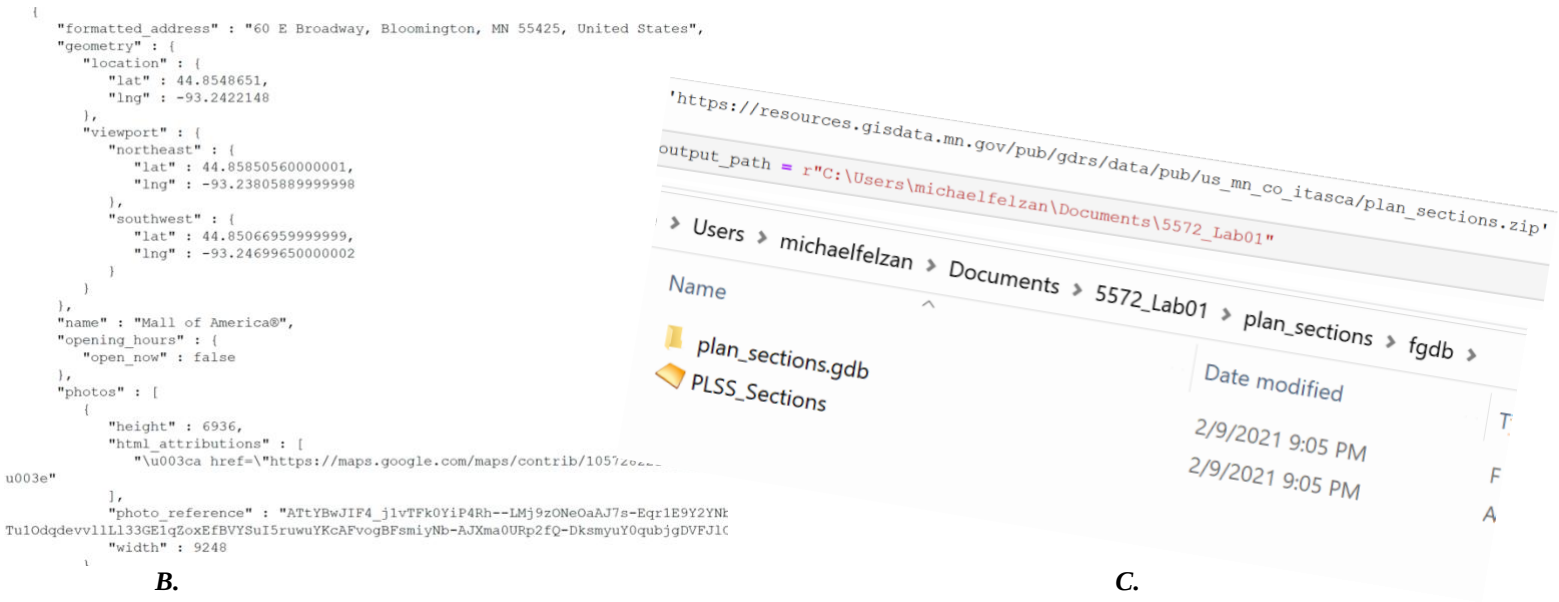
Fig. 14. Code demonstration of passing specific parameters into a GET request to the NDAWN api to receive a curated .CSV of weather data

As explained above, we may simply create a dictionary of key:value pairs, which correspond to the specific data we want, to be passed into our GET request as URL parameters, ultimately yielding a curated .csv file. However, once the request is received and sent back to be assigned to a variable as a request object, we need to decode the content (here we'll use 'utf-8,' 8 bit), and then use the pandas.read_csv function to access the data. The table shown at the end of ***Fig. 14.*** is the data, relating to the specific parameters I passed (station X, day Y) I wished to extract using the NDAWN API.

Results // Results Verification

	Station Name	Latitude	Longitude	Elevation	Year	Month	Avg Temp	Number Missing	Number Estimated	Monthly Normal Average Air Temperature	Departure from Normal Monthly Average Air Temperature
0	NaN	deg	deg	ft	NaN	NaN	Degrees F	NaN	NaN	Degrees F	Degrees F
1	Zeeland	46.013378	-99.687587	2070	2020.0	2.0	15.701	0.0	0.0	17.00	-1.299
2	Zeeland	46.013378	-99.687587	2070	2020.0	3.0	29.808	0.0	0.0	28.00	1.808
3	Zeeland	46.013378	-99.687587	2070	2020.0	4.0	37.843	0.0	0.0	43.00	-5.157
4	Zeeland	46.013378	-99.687587	2070	2020.0	5.0	52.073	0.0	0.0	55.00	-2.927

A.



B. ***C.***

Fig. 15. A: Code output/result of CKAN API ETL; B: result of Google Places API ETL; C: Result of NDAWN API ETL

The results of the ETL processes for each of the three APIs discussed in this report are visualized above. The methodology outlined in this report seems to be accurate, as I received non-corrupted data from each of the ETL protocols. Although extracting data via an API and saving it to your personal computer was a primary objective of this lab, I believe the most valuable product here are the protocols themselves, as their general frameworks may be reappropriated for various other applications

Discussion and Conclusion

To speak in a more personal tone, this lab was a major learning experience for me – I had never realized URLS contain parameters after the "?" in the link, which operate in the same way as those in a python function. I believe I was successful in building an ETL routine for each of the three APIs in an Esri Jupyter Notebook environment, and in the process, further solidified my understanding of how APIs (and the internet as a whole) work.

The three APIs discussed in this lab have their own subtly different rulesets that you have to abide to, though the general process of using computer programming to communicate with an interface to receive specific information back, is the same. The python "requests" module allows for the communication with each of the APIs covered in this lab, and the file architectures these APIs are connected to are similarly built (as it is how the whole web is built).

I'm sure a couple of these tasks would have been easier given the opportunity to use an html parser like Beautiful Soup, though it was a rewarding challenge to be limited to the Esri environment. In the process, I was able to utilize Chrome's developer tools and deconstruct how the elements comprising a website relate to each other. Having built this understanding, I was able to manipulate HTML as a text string to scrape relevant data from the web. Additionally, I have further solidified my understanding what the Python requests library actually "does", the difference between GET and POST, and how to decode responses to extract abstract information (1s and 0s) into usable data.

References

(1) https://developers.google.com/places/web-service/get-api-key

(2) https://developers.google.com/places/web-service/overview

(3) https://docs.ckan.org/en/ckan-2.1.5/api.html

(4) https://gettecr.github.io/noaa-api.html#.YCQlxhNKhZp

(5) https://searchapparchitecture.techtarget.com/definition/RESTful-API#:~:text=A%20RESTful%20API%20is%20an,deleting%20of%20operations%20concerning%20resources

(6) https://www.searchenginejournal.com/technical-seo/url-parameter-handling/#close

Self-score

Category	Description	Points Possible	Score
Structural Elements	All elements of a lab report are included **(2 points each)**: Title, Notice: Dr. Bryan Runck, Author, Project Repository, Date, Abstract, Problem Statement, Input Data w/ tables, Methods w/ Data, Flow Diagrams, Results, Results Verification, Discussion and Conclusion, References in common format, Self-score	28	**26 (I know my references are lacking**
Clarity of Content	Each element above is executed at a professional level so that someone can understand the goal, data, methods, results, and their validity and implications in a 5 minute reading at a cursory-level, and in a 30 minute meeting at a deep level **(12 points)**. There is a clear connection from data to results to discussion and conclusion **(12 points)**.	24	**23 (hope I didn't blather)**
Reproducibility	Results are completely reproducible by someone with basic GIS training. There is no ambiguity in data flow or rationale for data operations. Every step is documented and justified.	28	28
Verification	Results are correct in that they have been verified in comparison to some standard. The standard is clearly stated **(10 points)**, the method of comparison is clearly stated **(5 points)**, and the result of verification is clearly stated **(5 points)**.	20	**20**
		100	**97**