Workshop 3: Application Programming Interfaces (APIs)

Myles D. Garvey, Ph.D

Winter, 2020

1 Learning Objectives

1. To understand the basic structure of any API.
2. To learn how to use any API by understanding the structure of its documentation.
3. Illustrate the use of data gathering via APIs.
4. Understand how a basic client-server model works with Request-Response protocols.
5. Use two example APIs (Twitter and Yelp).

2 Workshop Description and Submission

In the first workshop, you learned how to write (and use) your own functions to return information. As it turns out, many organizations also have libraries of functions of their own that allow for the public to use for the goal of retrieving information. Such a collection of functions is called an *Application Programming Interface*. As the name suggests, these software products are intended for software developers and analyst using a programming language to automatically search, download, and even post information programmaticly.

It is of no doubt that knowing how to learn an API, as well as integrate it in your own analysis, can greatly expand your access to data, as well as your ability to quickly download and analyze it. In this workshop, we will demonstrate how a basic API works, how to learn the fundamentals of a specific API, and how to interact with an API both through a tool and code.

You will submit to blackboard a .PDF document. Complete each exercise in this workshop. These are easy and small, and so it should not take you very long. In a word document, type up the question number, the original question text, and your response in R code. Save your word document as a .PDF and submit it to the submission link on Blackboard. Please note that failure to submit this as a .PDF will result in a 0 on the workshop.

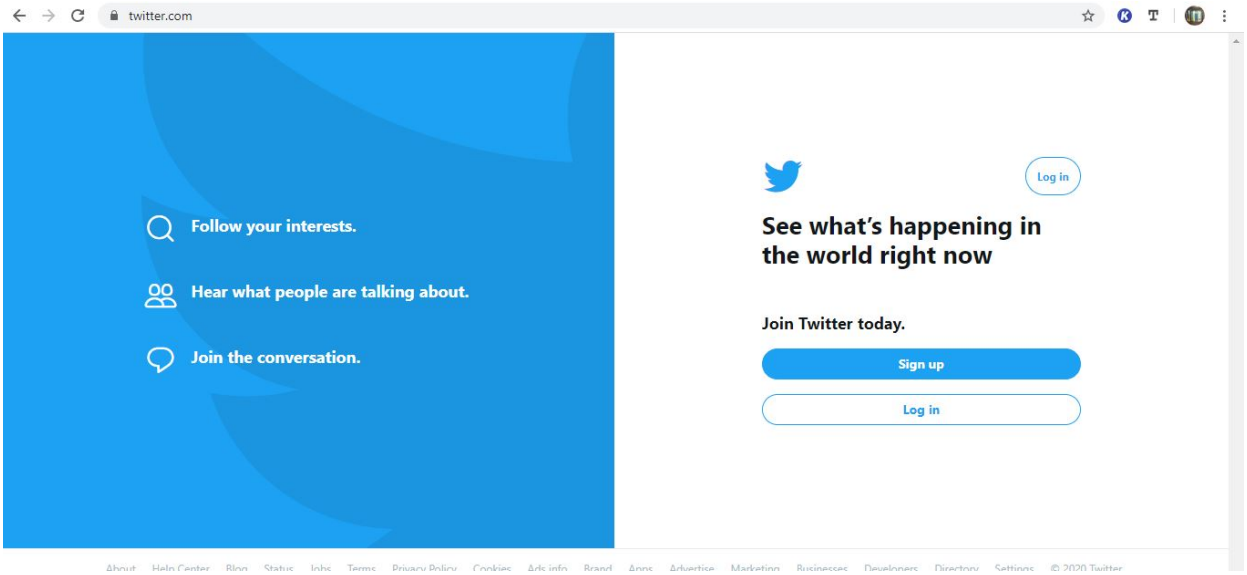
3 Installation

Before we get into the workshop, you will need to accomplish a few tasks, namely:

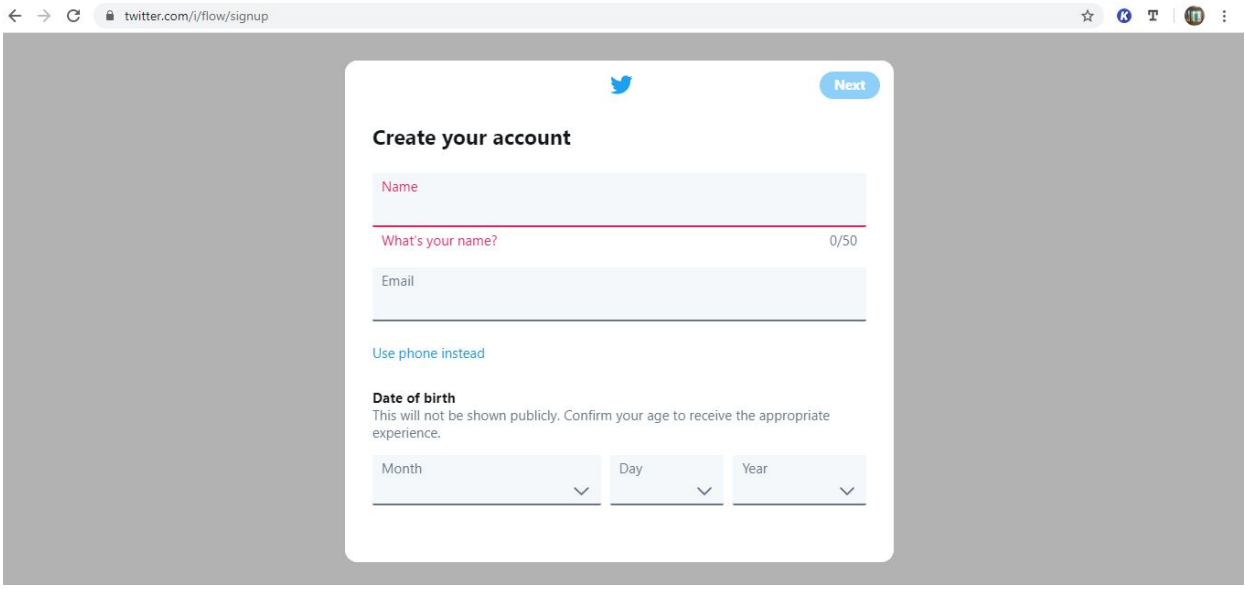
- Set up a Twitter user and developer account.
- Set up a Yelp! Account.
- Install a REST Client (We will use Insomnia)

3.1 Setting up a Twitter Developer Account

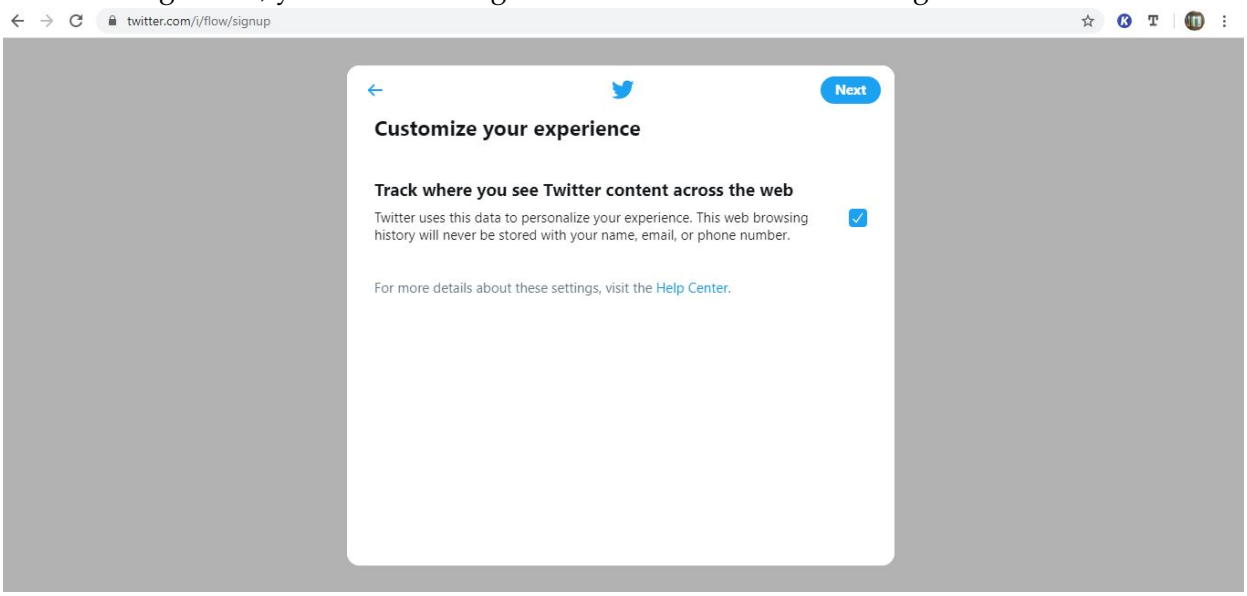
First, you will need to set up a regular plain old Twitter account. I can understand some people's pensiveness to do so. However, it is mandatory for this course. You are free to delete the account after the completion of this course. With this stated, lets get started. First, browse to http://twitter.com. If you do not already have an account, and you are not logged in, it will display the screen below. Click "Sign Up":



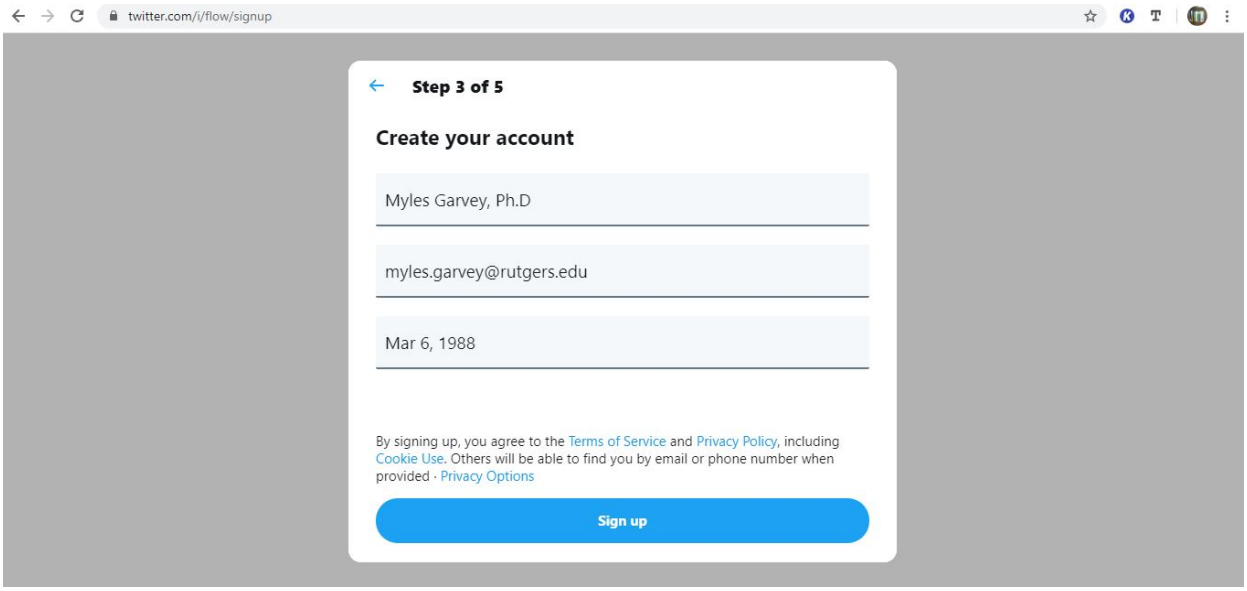
You can create an account using either your email or your phone. For developer accounts, **you will need both**, so ensure you have access to a phone number that allows for text message verification. You can start signing up by email by first clicking "Use email". Next, put in your birth date:



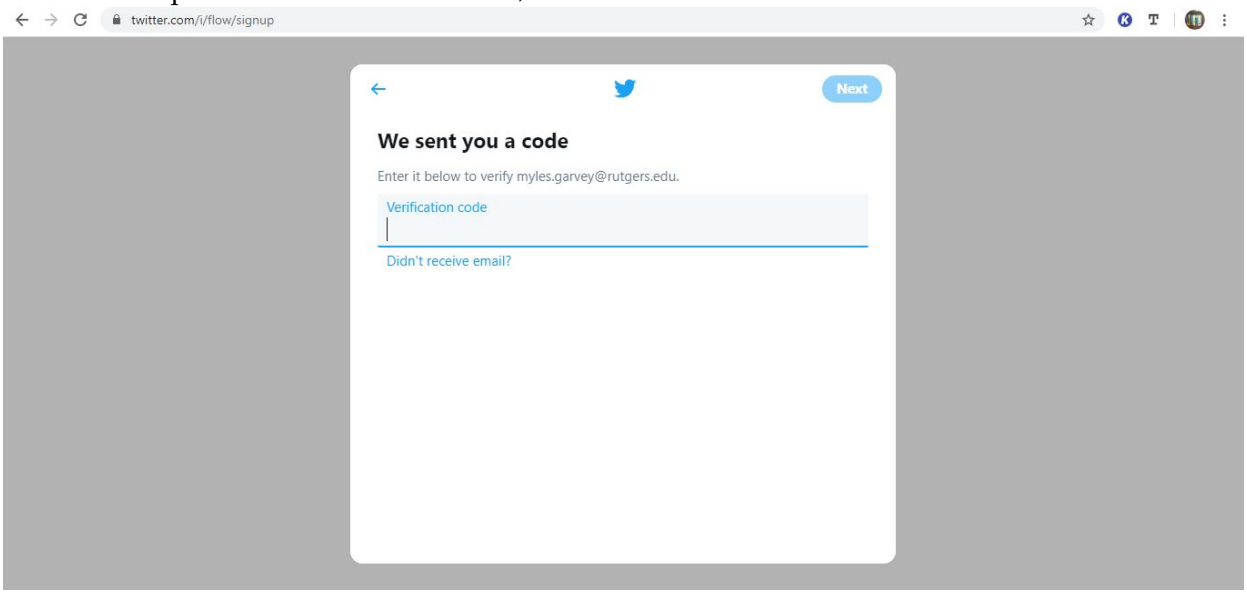
After clicking "next", you will be brought to this screen. Click "Next" again.



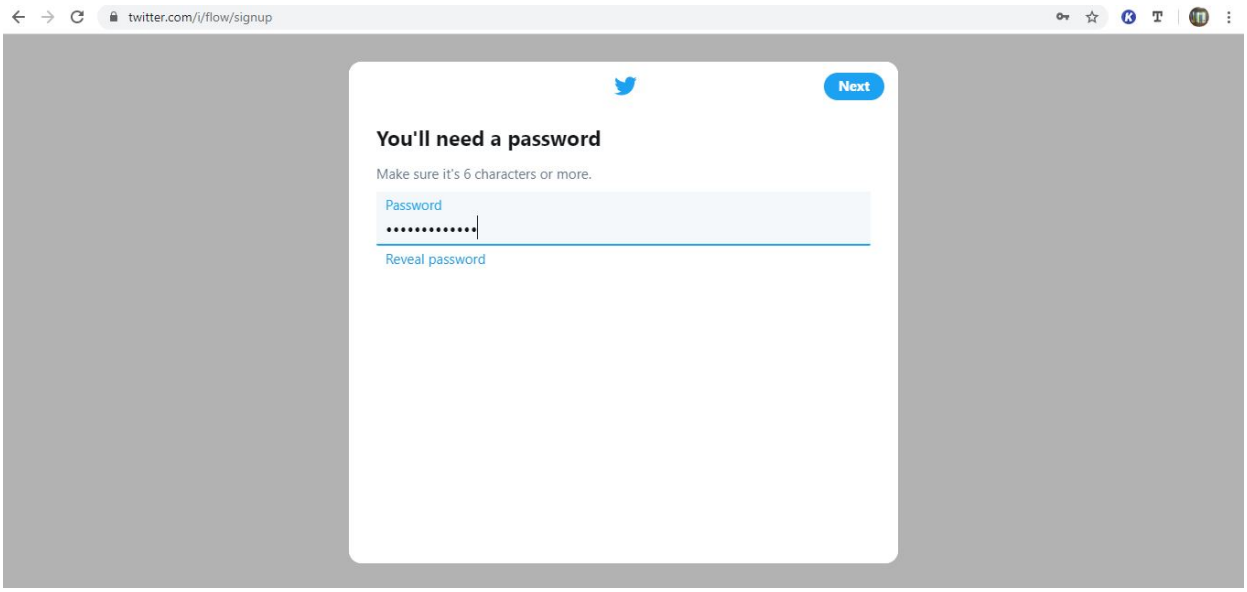
Now verify your details, and click "Sign Up":



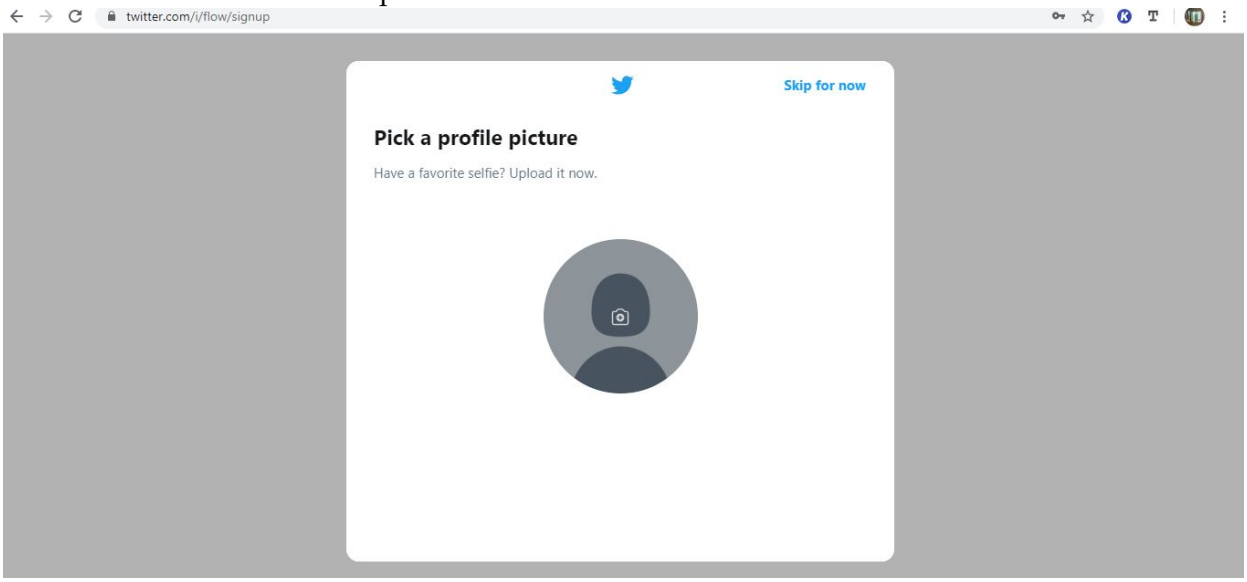
Twitter will now send a code to your email/phone (depending on how you signed up). Get the number and put it in the text box. After, click "Next":



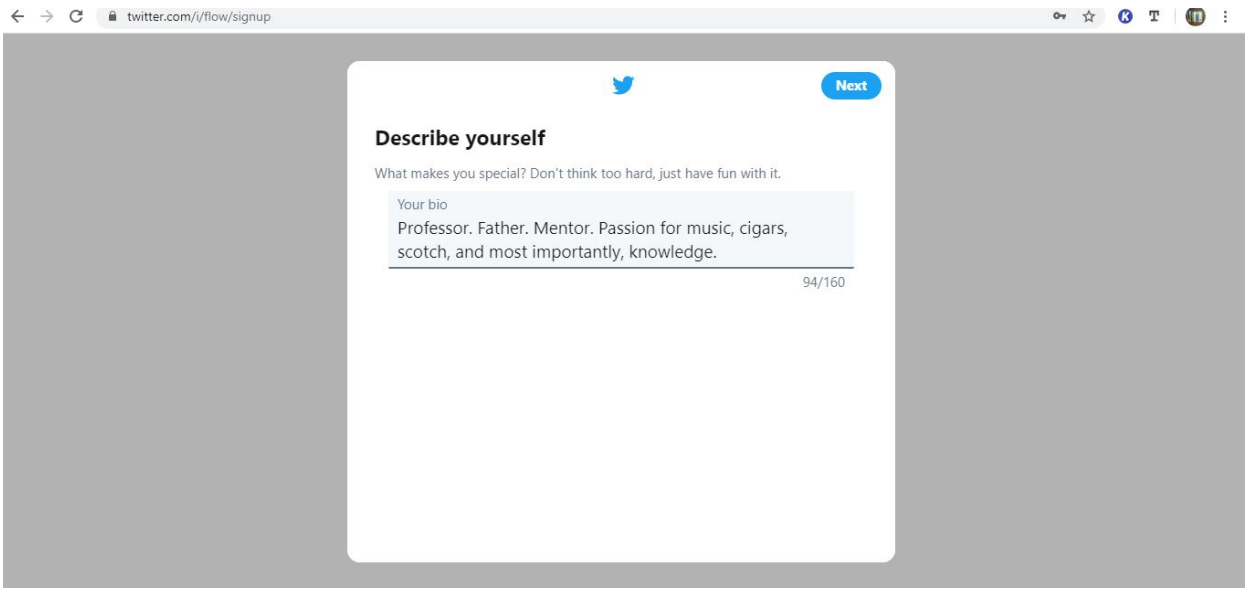
Now create a password and click "Next":



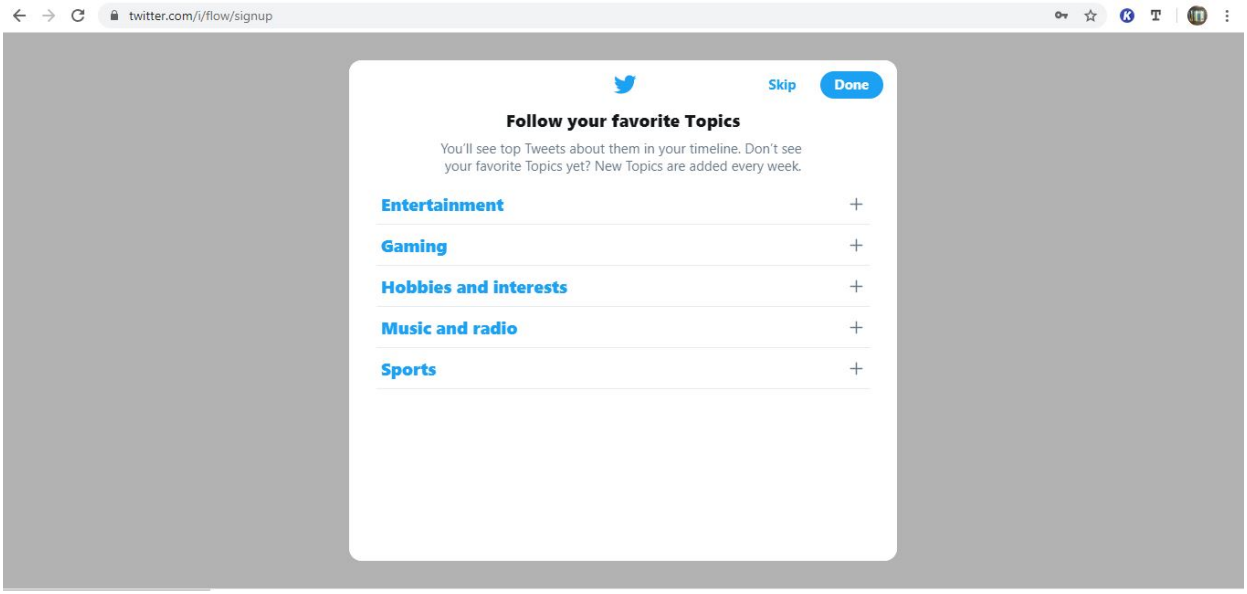
You don't need a profile picture if you do not want one. Click "Skip for now", the remaining details are not needed for developer account.



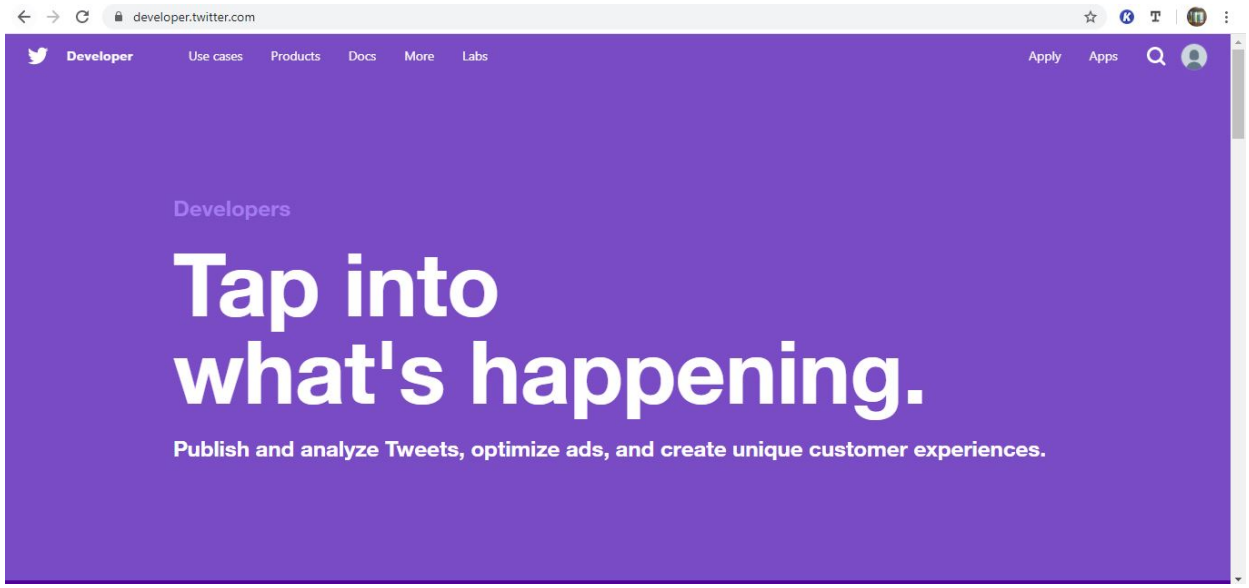
Now take a quick moment to describe yourself (this will be public info, just keep that in mind). I forget if Twitter allows you to keep it blank. If not, then just write nonsense.



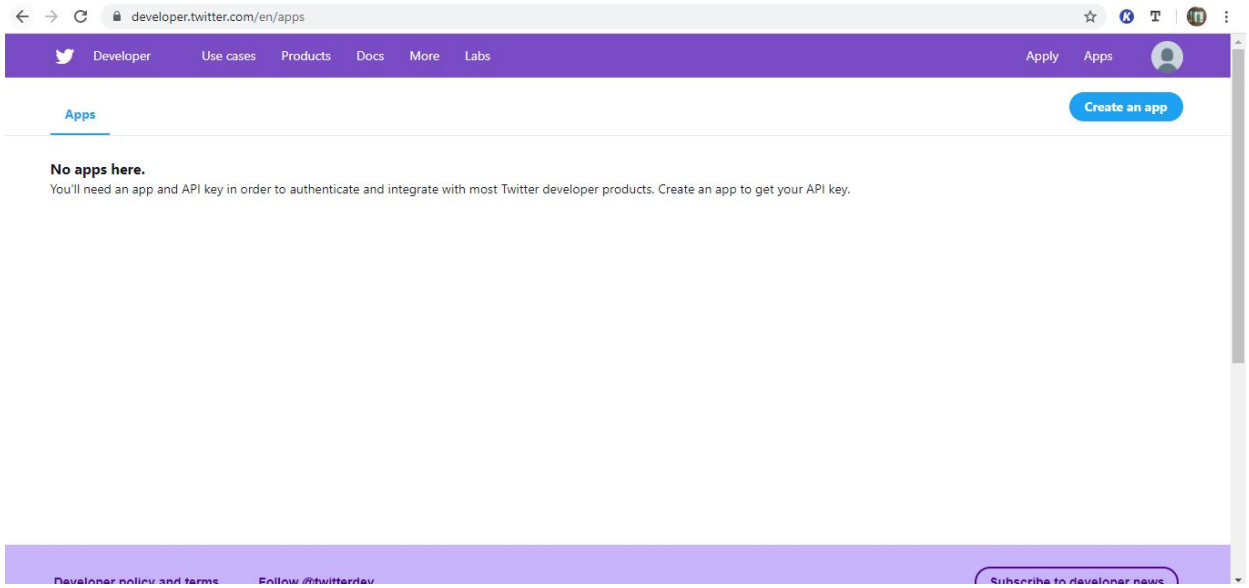
Now the rest is just personalising of your Twitter. Just ignore, and click "Done".



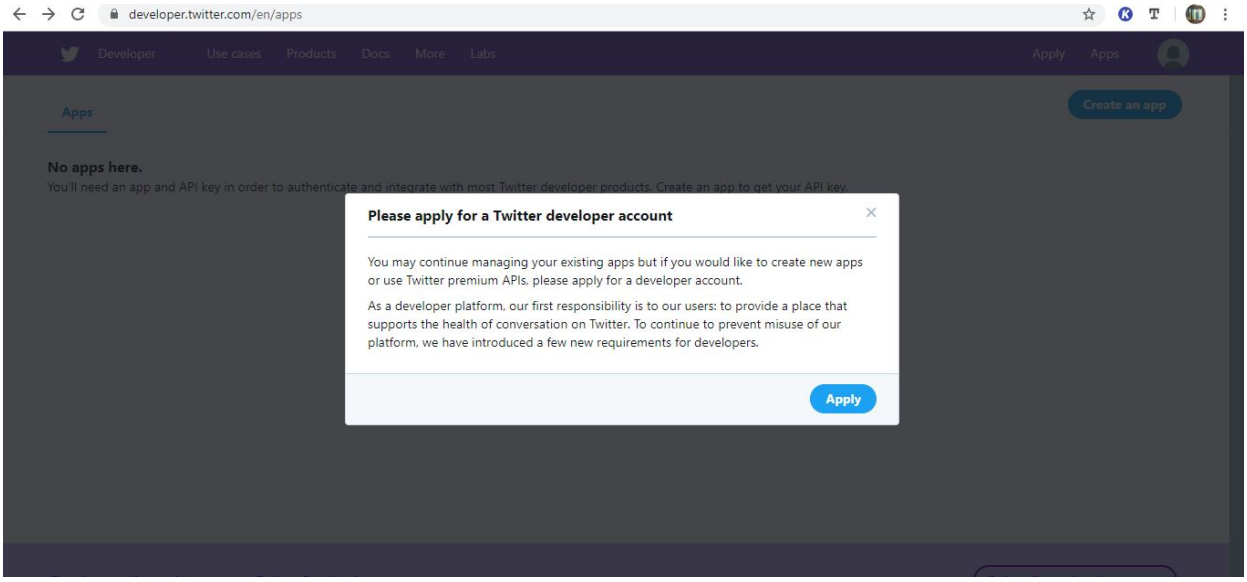
Now we will sign up for a developer account. Browse to http://developer.twitter.com:



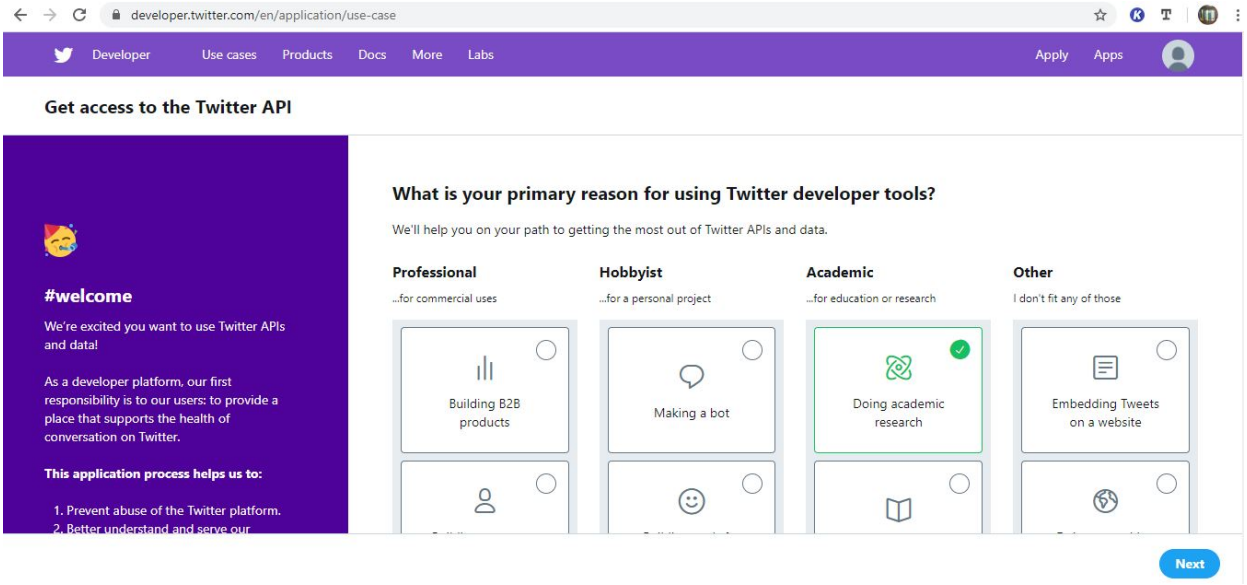
Now click "Apps", which will bring you here:



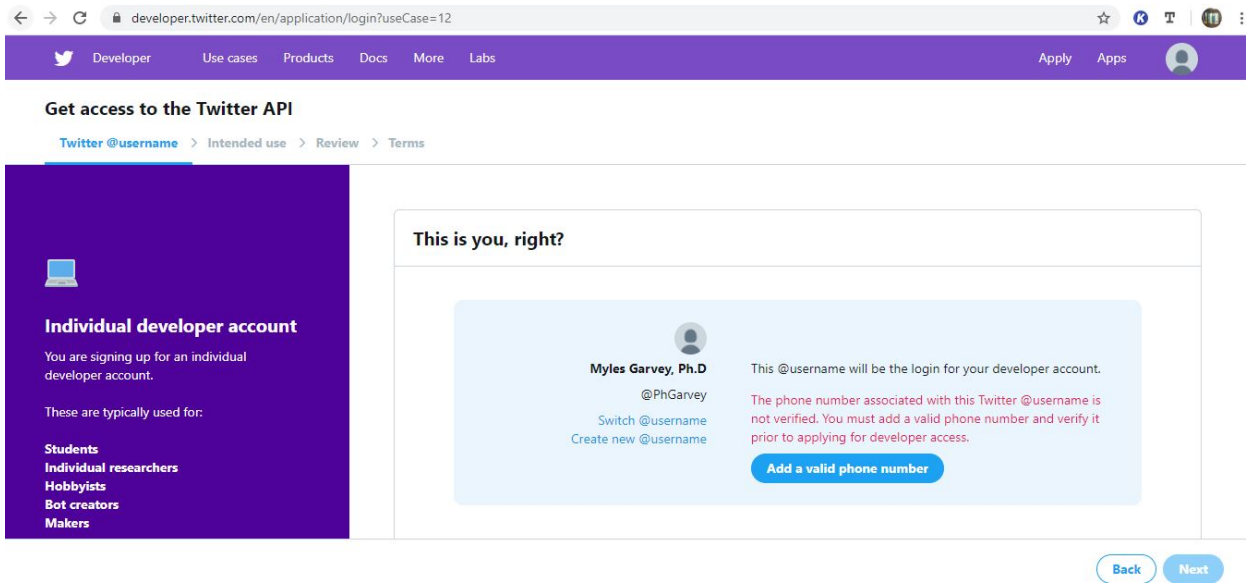
Click on "Create an app", which will now ask you to apply for an account:



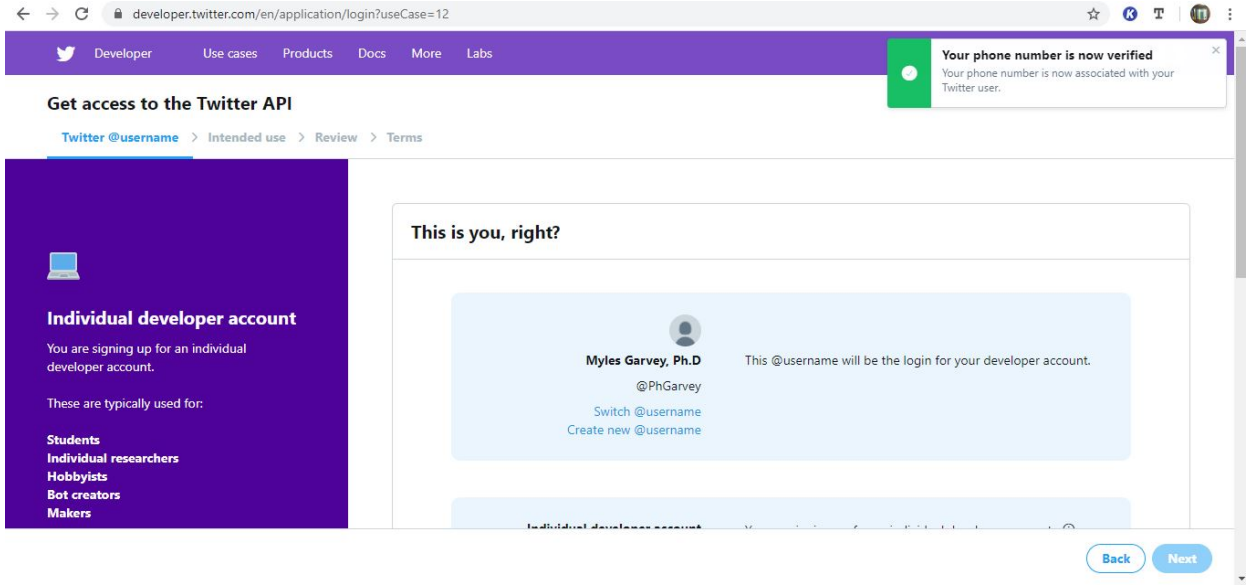
Click "Apply":



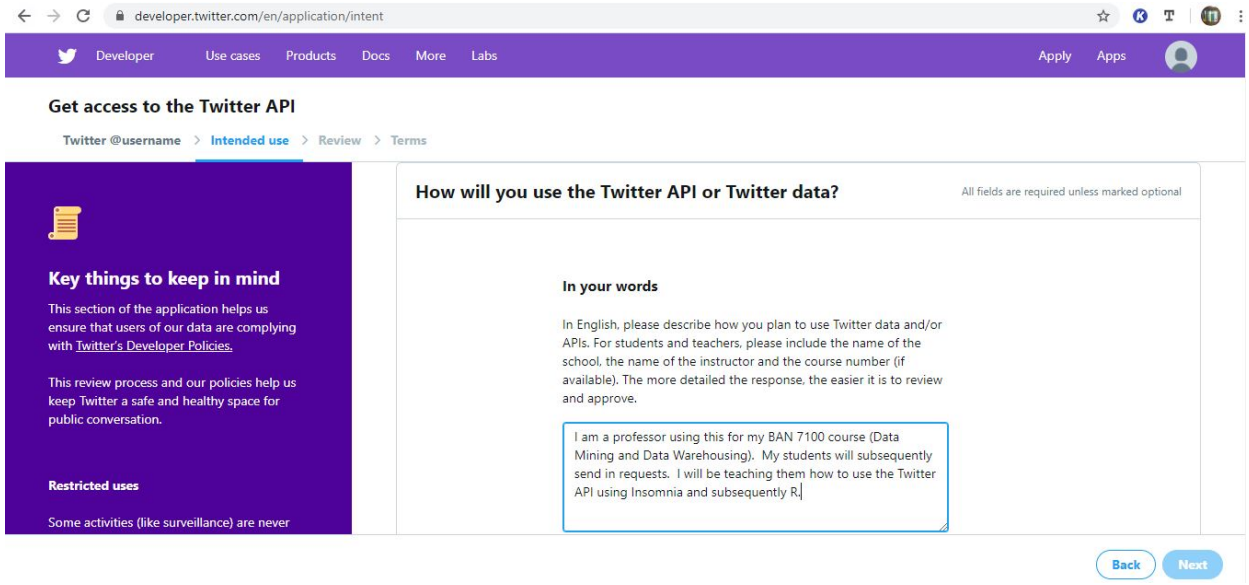
Give a reason for applying. We are doing this for academic research, so I just clicked that. Click "Next", which will bring you here:



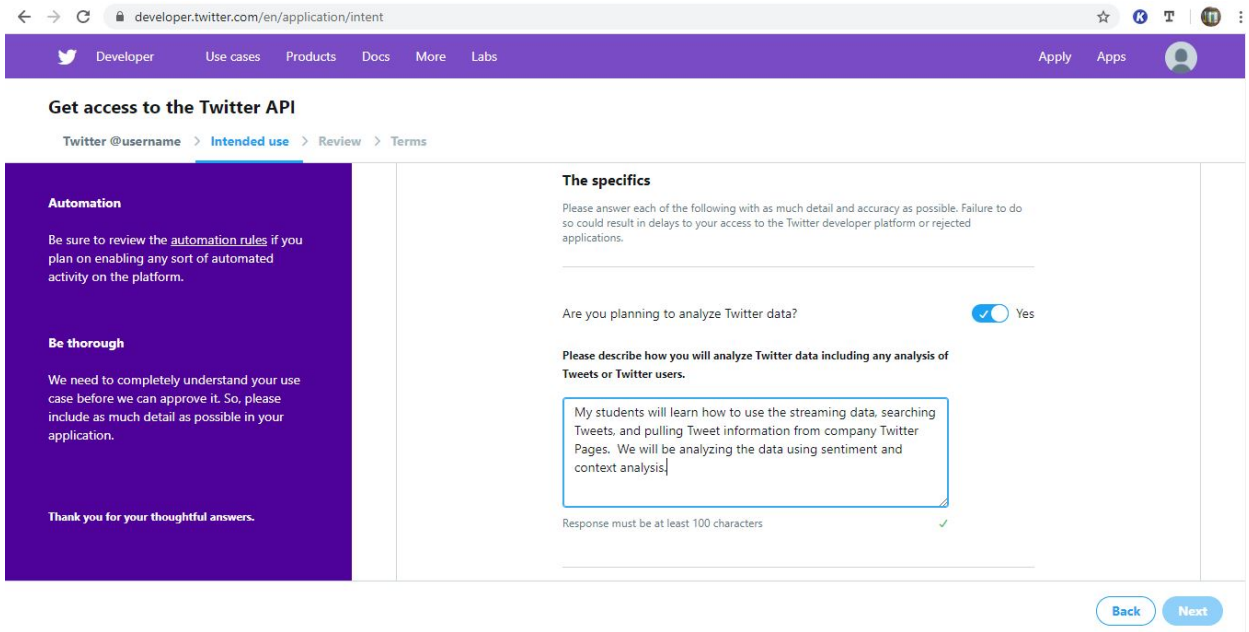
Now Twitter will ask you to verify your email and phone number, if you have not already done so. Click "Add a valid phone number". It will then ask your for your phone number to add. Add it in, and verify as usual via the phone. Once verified, you should see this:



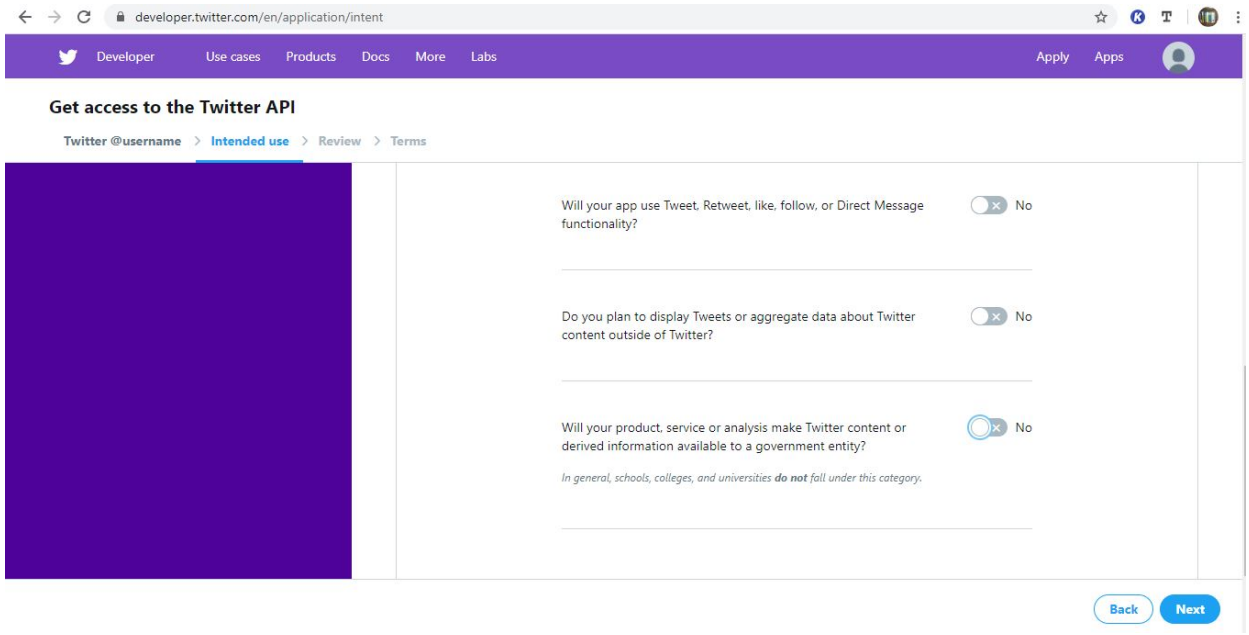
Twitter will now ask you about how you intend to use the developer account. Fill in all (Required) in this form:



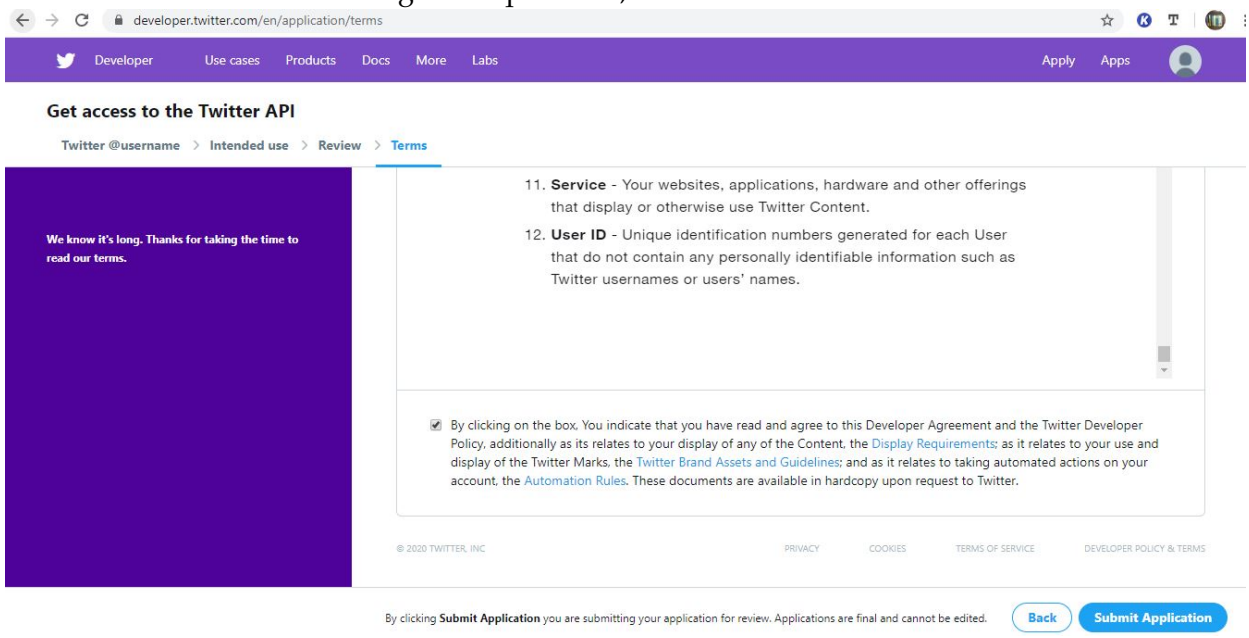
Scroll down and fill in additional details. Twitter will ask you what you're using it for. Just say you are enrolled in my class and that you are using it for coursework. It will ask a minimum number of characters. Expand on how you may use it just to get to the character minimum. Don't add non-sense, or your account may be delayed for approval:



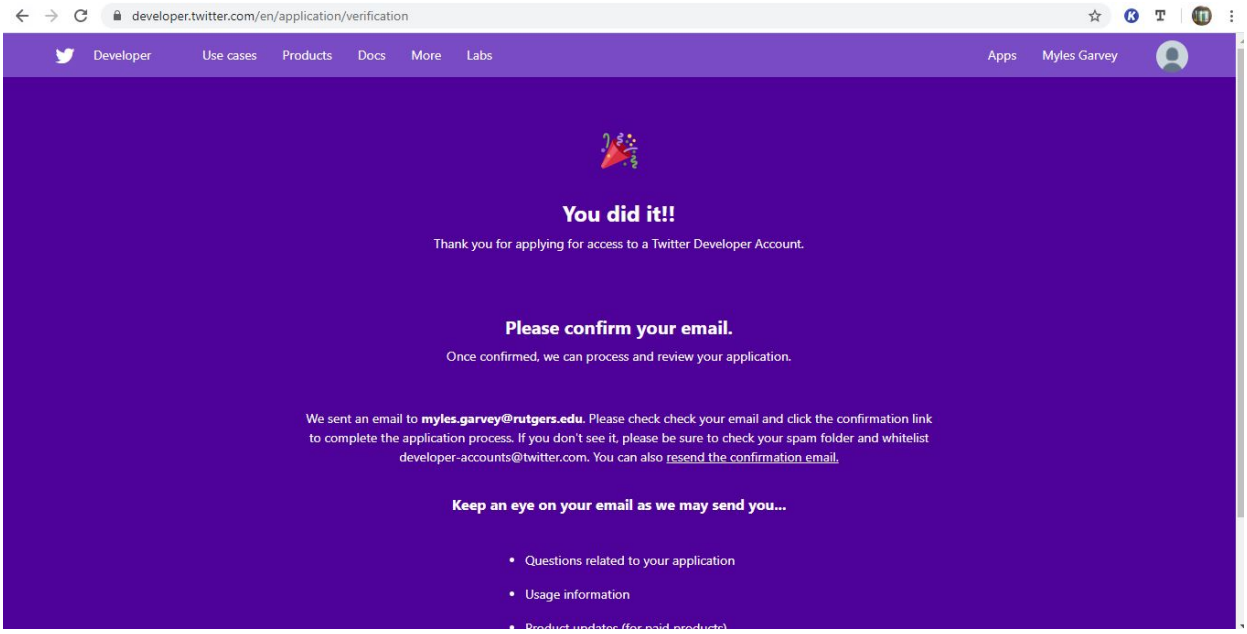
Now ensure you click that you are analyzing data. You can copy my description and use it in yours:



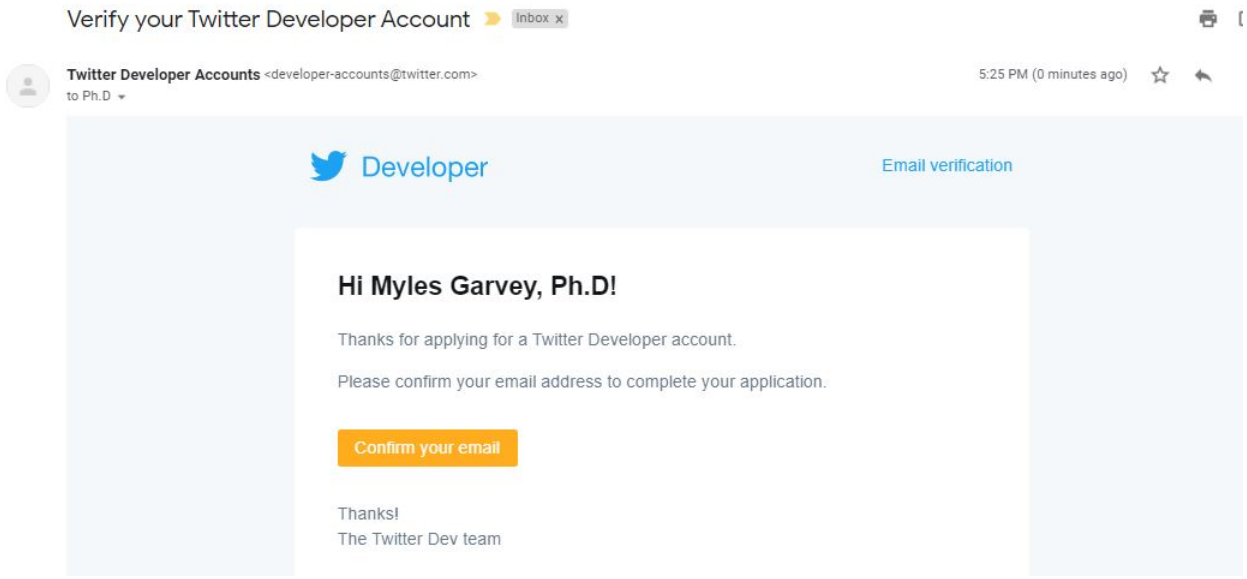
Click on "No" for the remaining three questions, then click "Next":



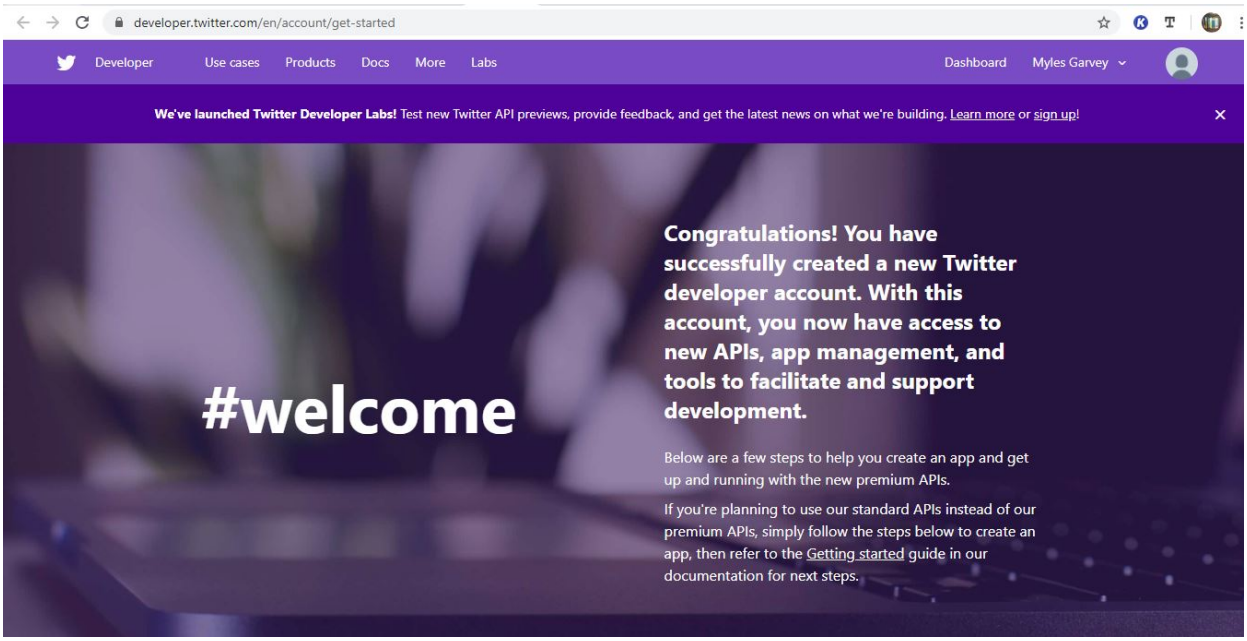
Read the terms of use and service, click the check box, and click "Submit Application":



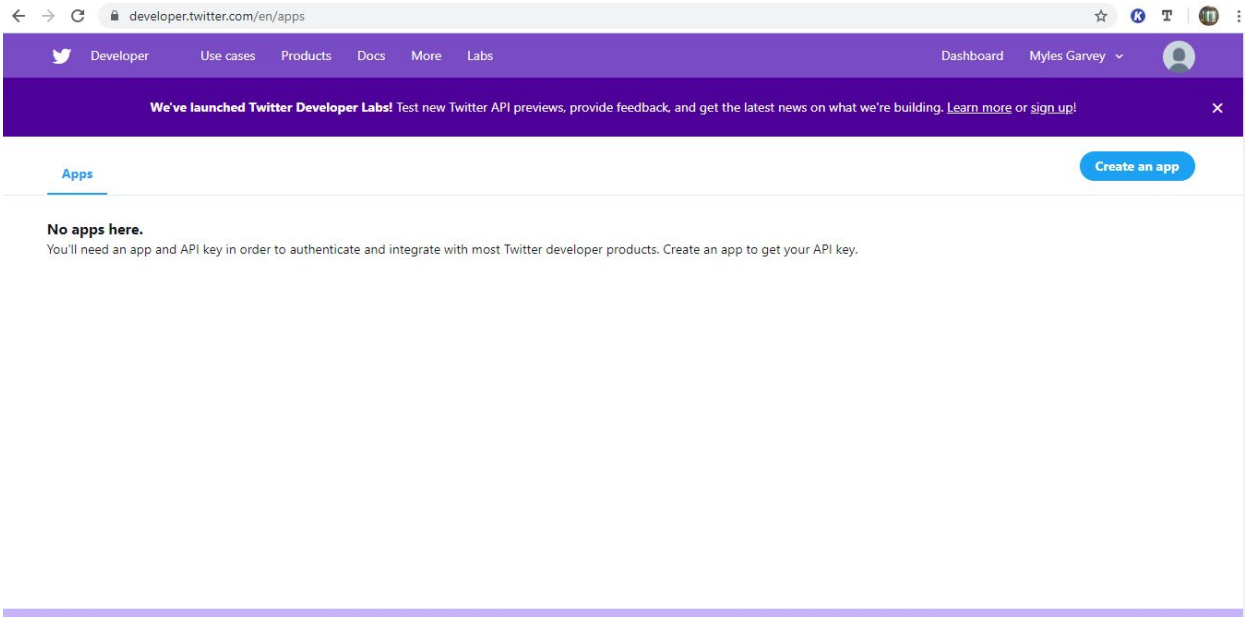
Congrats! You are now done with the developer account. You are now asked to verify via email:



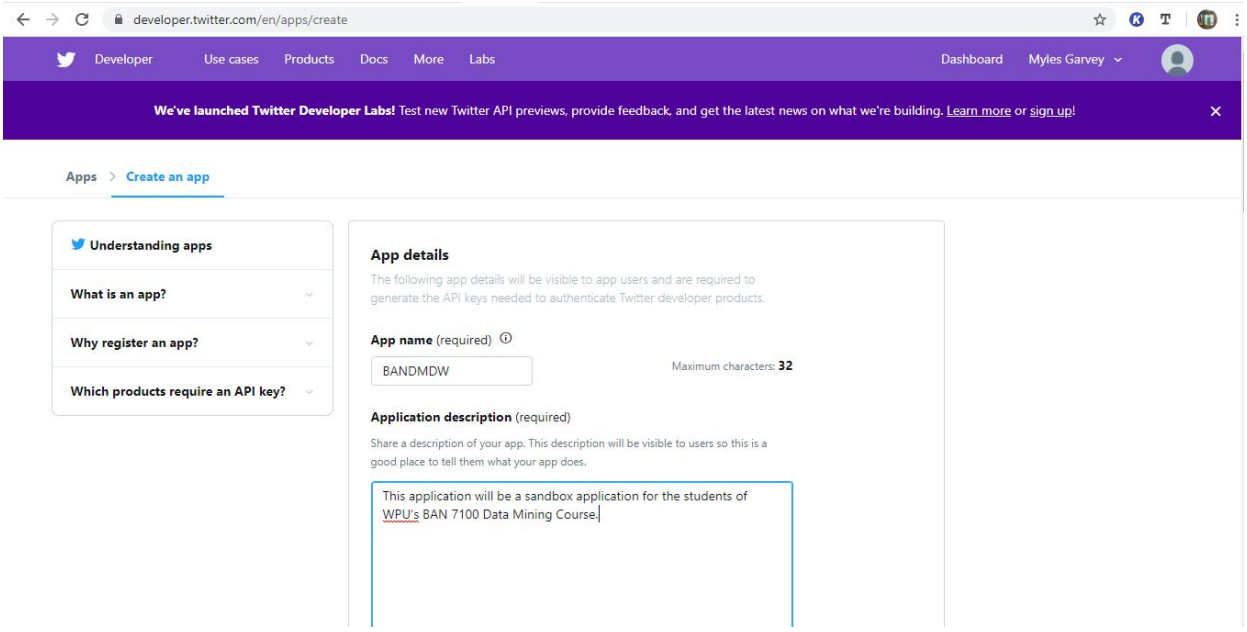
Check your email, and once inside, click "Confirm your email":



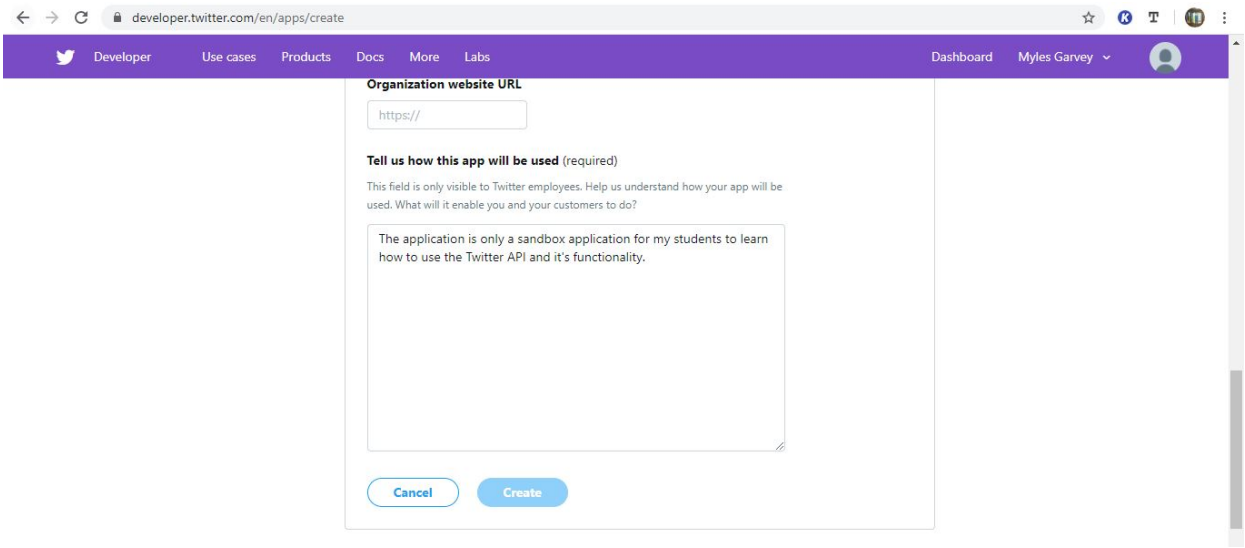
Congrats, you have now been verified. You should now go back to http://developer.twitter.com. Now click the drop down box on your name, and click "Apps". It should bring you here:

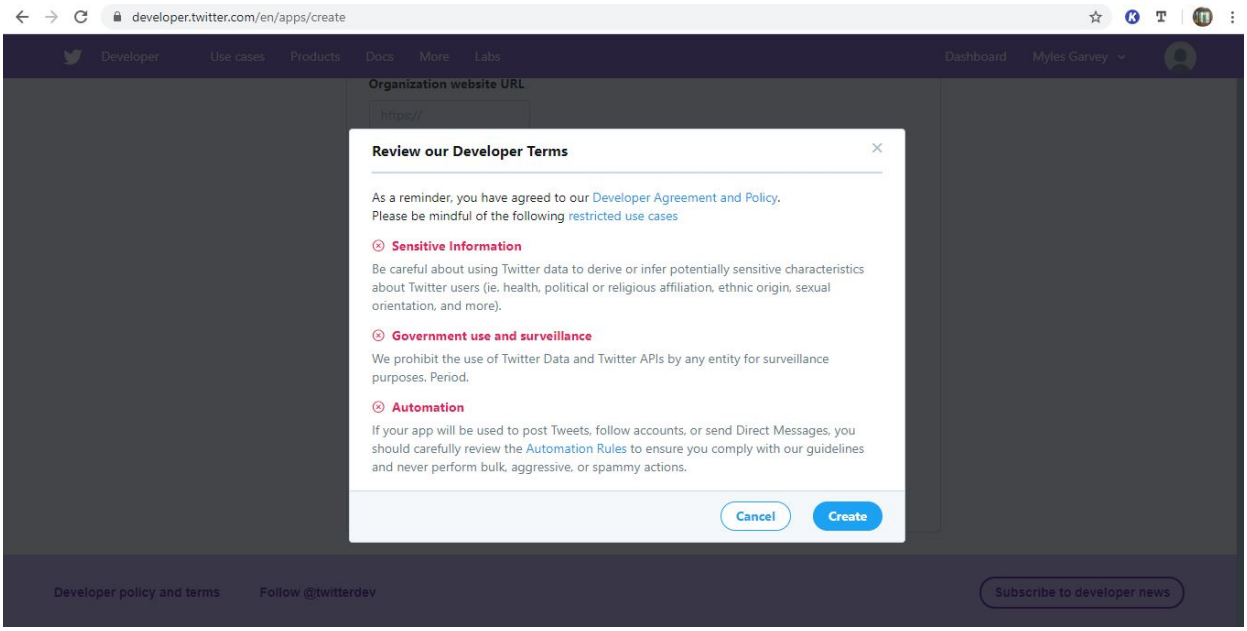


Click on "Create an app". It will now ask you to input an app name. Pick a name (any name). Hint: pick one without any spaces. All apps publicly on Twitter need to be unique, so you may need to use some numbers to make it more distinct. Also, tell twitter how you intend to use the app. You can copy my description if need be:

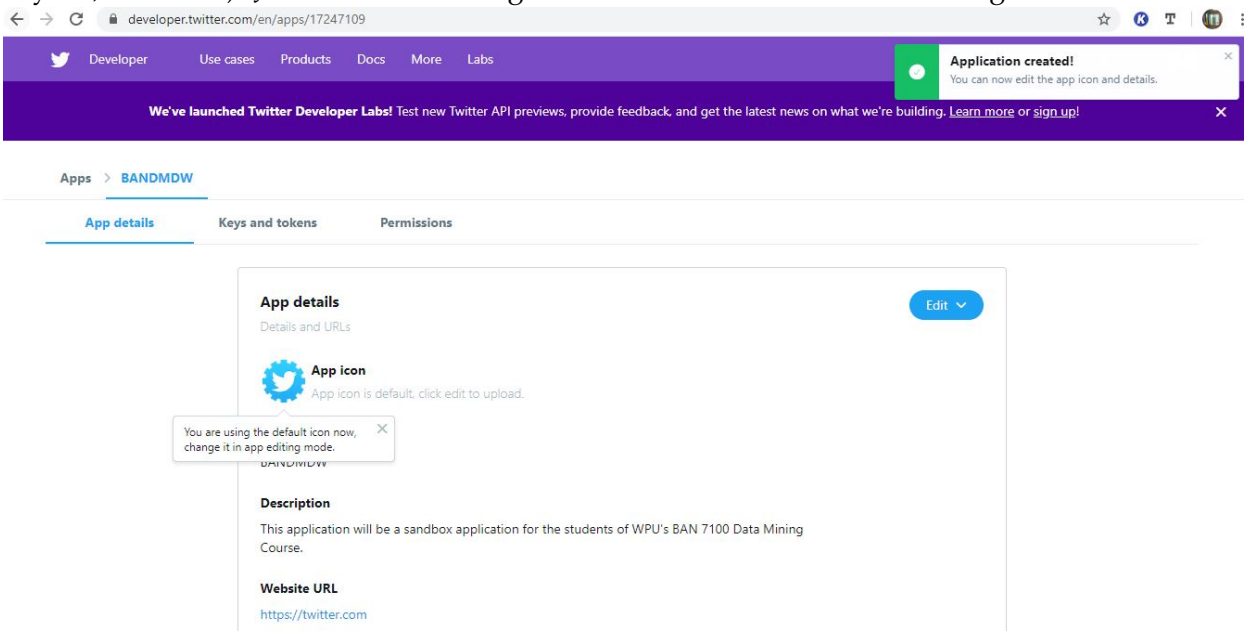


You can skip all other details, except those that say (Required). For the required URL, you can just put http://twitter.com. Last, you can copy my description on how the app will be used (or just come up with your own). When done, click "Create". If it is not letting you, then it means you missed a Required input:

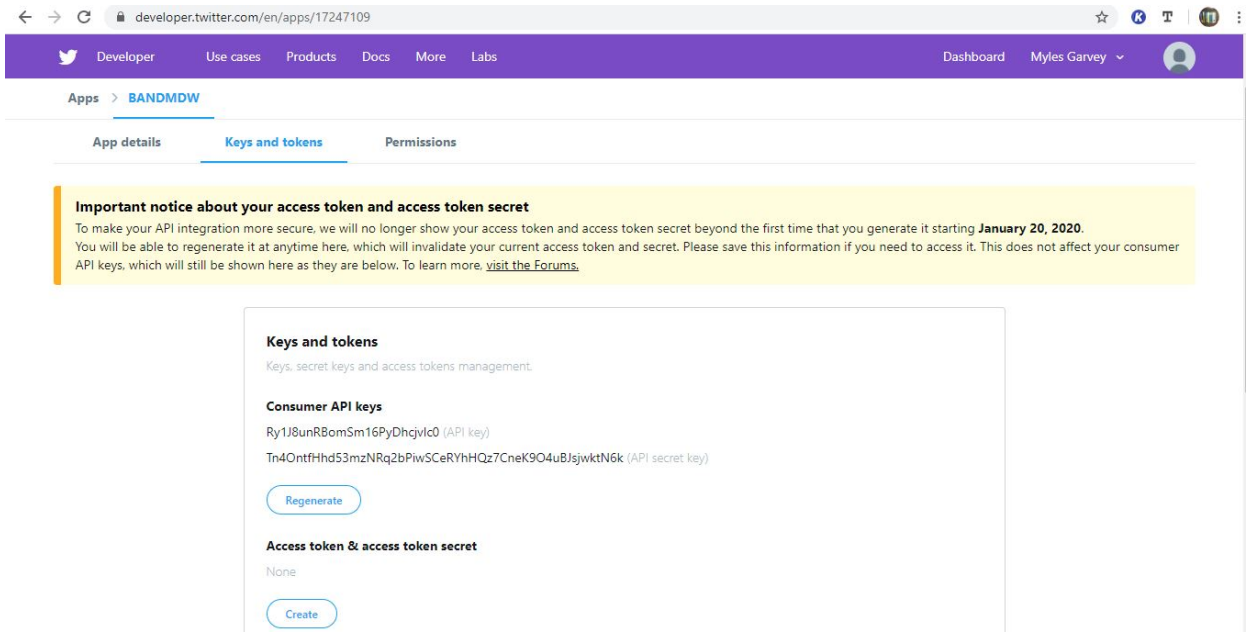




After clicking, you will be warned again as to what you are not allowed to do with the data (yeah yeah, we know). Just click "Create" again on this screen. You are now bought to here:

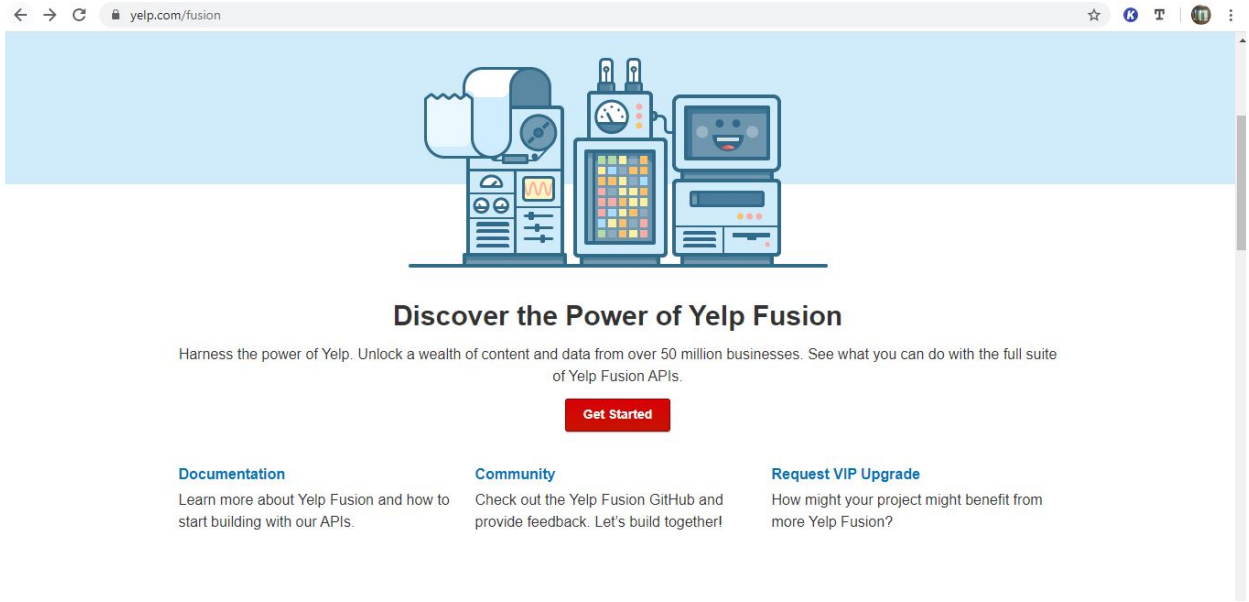


The last step is to get your keys and tokens, which are ways to access the Twitter API. Click on "Keys and tokens". The very last step is to generate an access token and secret. Click on "create" to generate this. IMPORTANT: copy and pact the Consumer API Keys and the Access tokens and secret to a text file, and save it for future use. As Twitter indicates, once you generate these, its one time view only. If you loose it, you WILL need to regenerate everything:

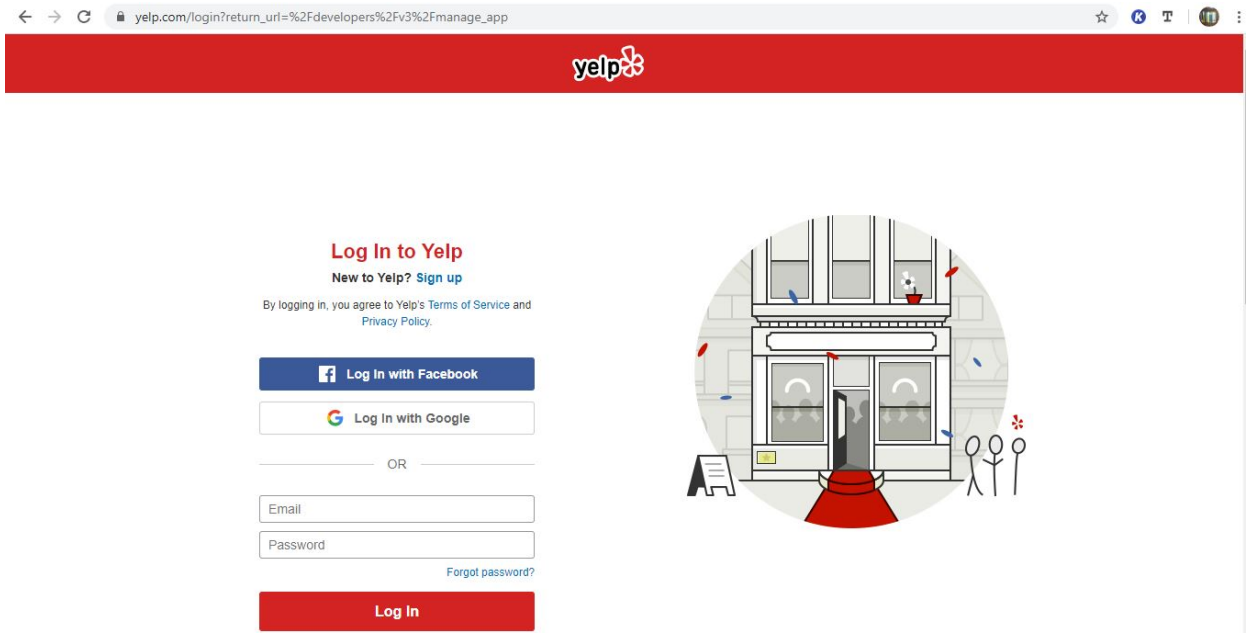


3.2 Setting up a Yelp! Developer Account

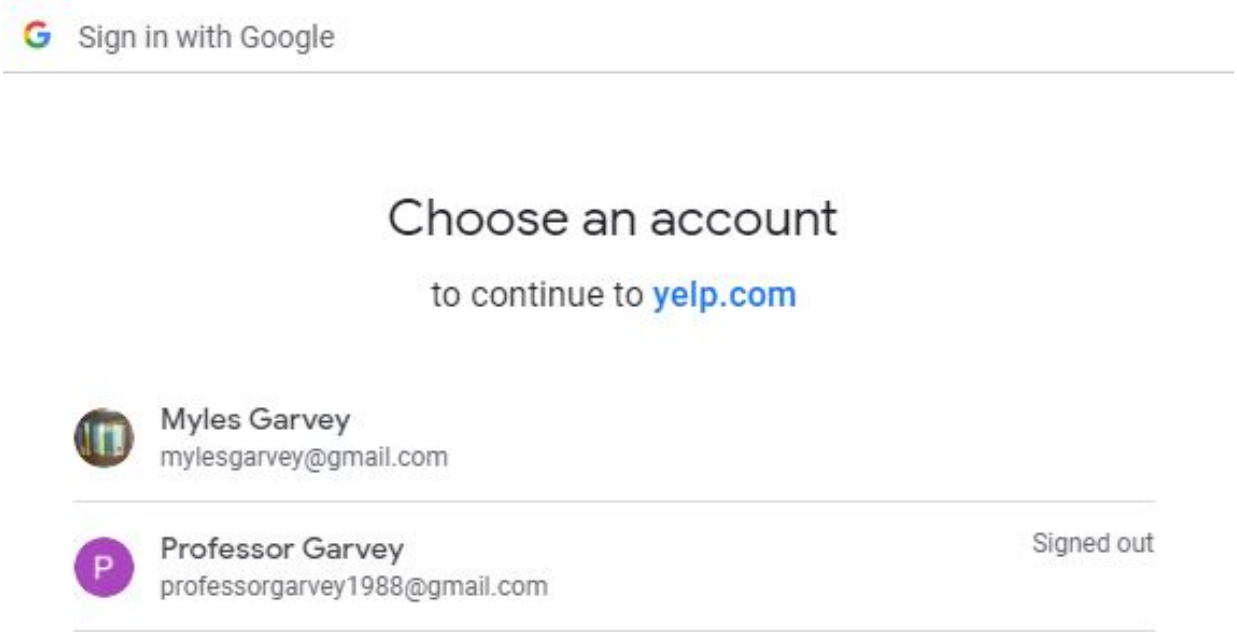
Since we will also be explored the Yelp! API, you will also need a developer account here. Luckily, this process is not as lengthy as that of Twitter. To begin, browse to https://www.yelp.com/fusion, scroll down, and click on "Get Started":



Now you have a few options here. The easiest (which is what I recommend, since it is the most secure), is to sign up via Google, but you are free to sign up however you wish. If following me, click on "Log in with Google":

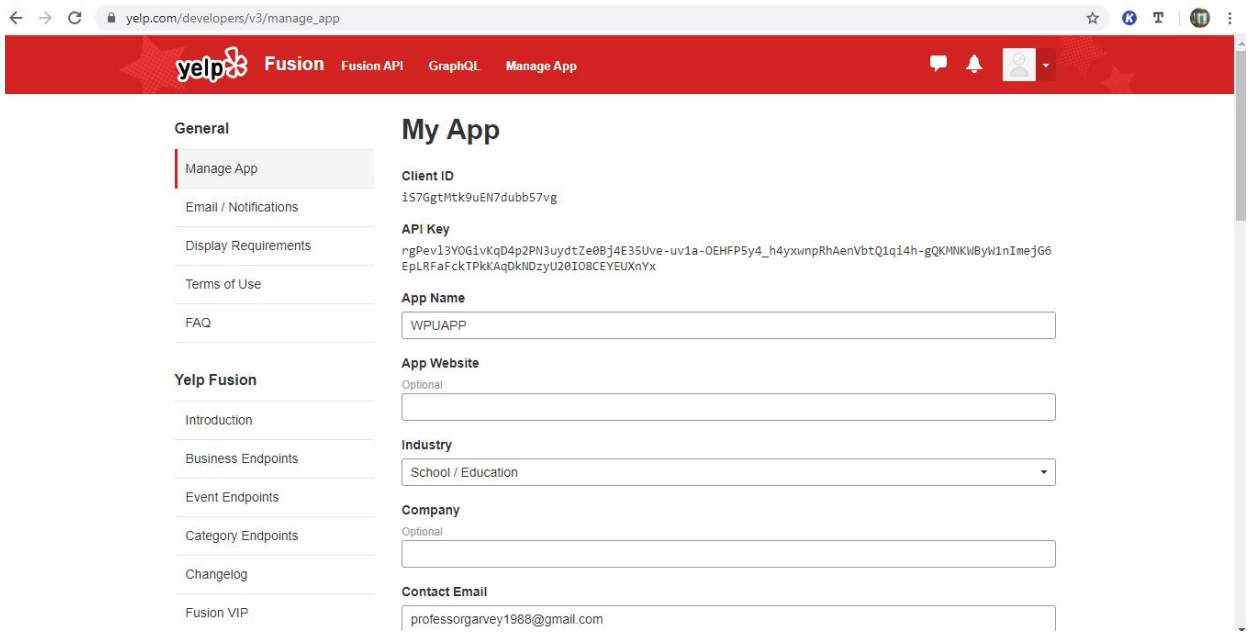


You will then be asked to select your Google Account, if you are already logged into one, like I am. If not, it will ask you to put in your gmail for google:

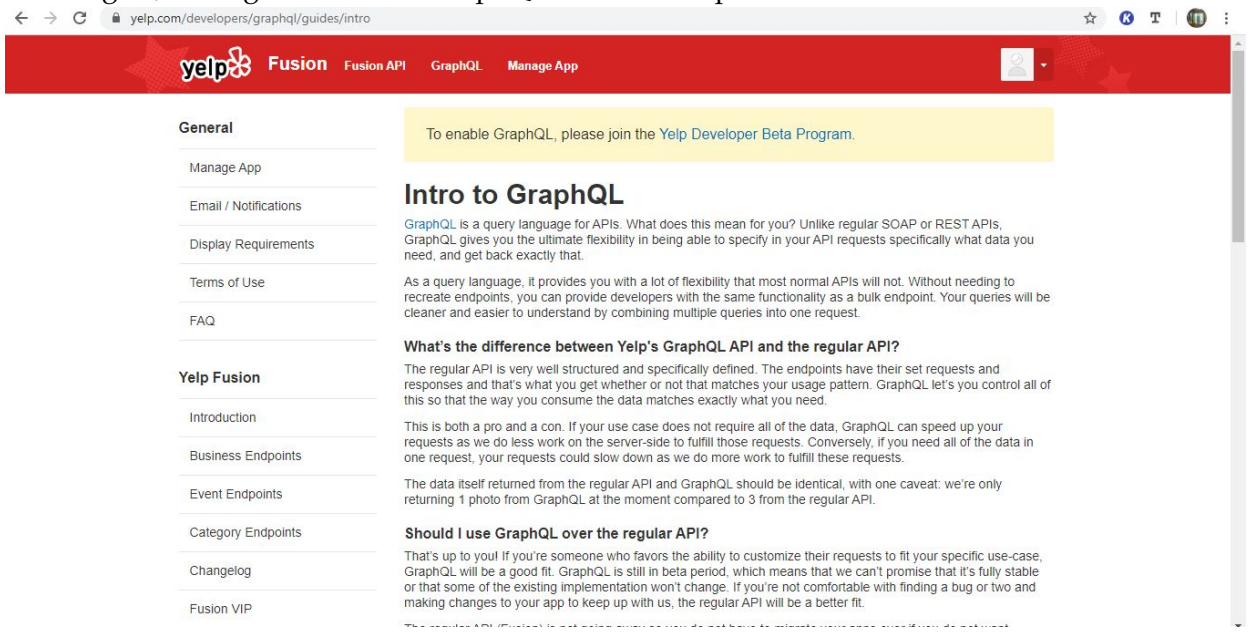


After clicking on your account, you will input your Gmail password (don't worry, this is a very secure method of signing up for a website). After doing so, you will be brought to this screen below. It is a similar idea that Twitter had with creating an "App". Fill in all required details, write a long enough description, and then click on "Save Changes". Like Twitter, ensure to copy and paste the "Client ID" and the "API Key" into a text file and save it for future reference!:

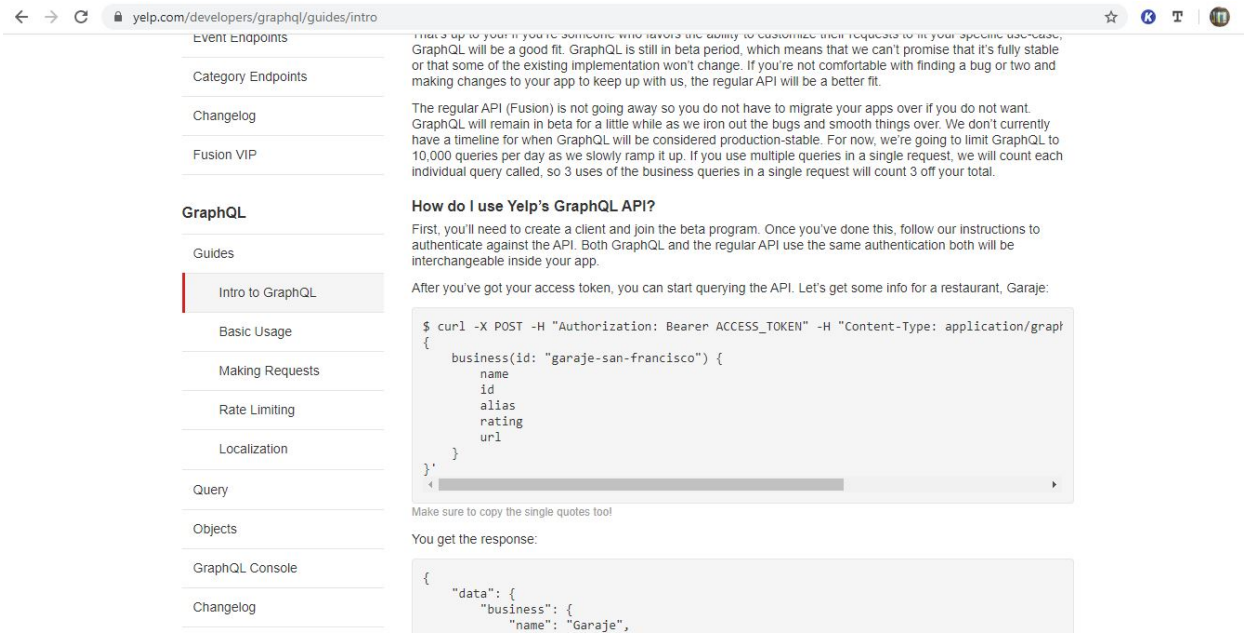
Now, click on "GraphQL" up top, which will bring you to this screen:



Click on "Yelp Developer Beta Program" (the link in the yellowish box on top). This will bring you back to the app screen. Scroll down, and click "Join Developer Beta". After doing so, click "Save Changes", and again click on "GraphQL" from the top:



The yellow box on the top should have disappeared (if it did not, you did not join the program correctly, and will need to attempt again):



In the screen for Graph QL, on the left menu, click on "Intro to GraphQL", and scroll down to "How do I use Yelp's GraphQL API?". You will see this text:

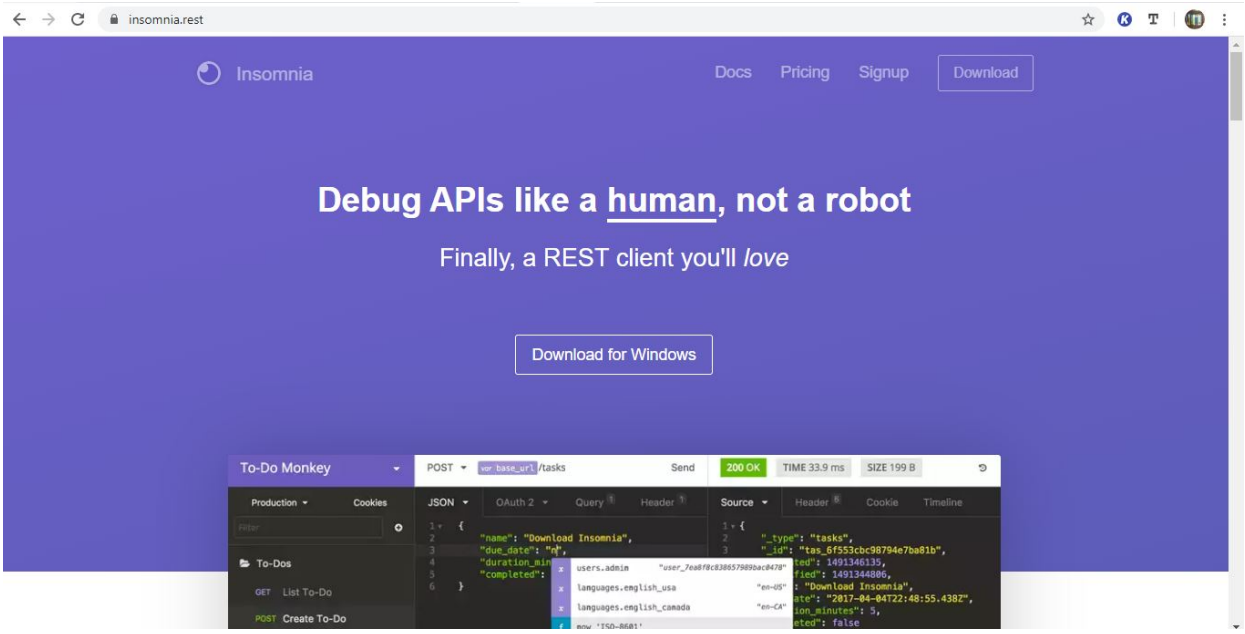
```
$ curl -X POST -H "Authorization: Bearer ACCESS_TOKEN" -H "Content-Type:
    application/graphql" https://api.yelp.com/v3/graphql --data '
{
    business(id: "garaje-san-francisco") {
        name
        id
        alias
        rating
        url
    }
}'
```

Listing 1: How to write a request to the Yelp database.

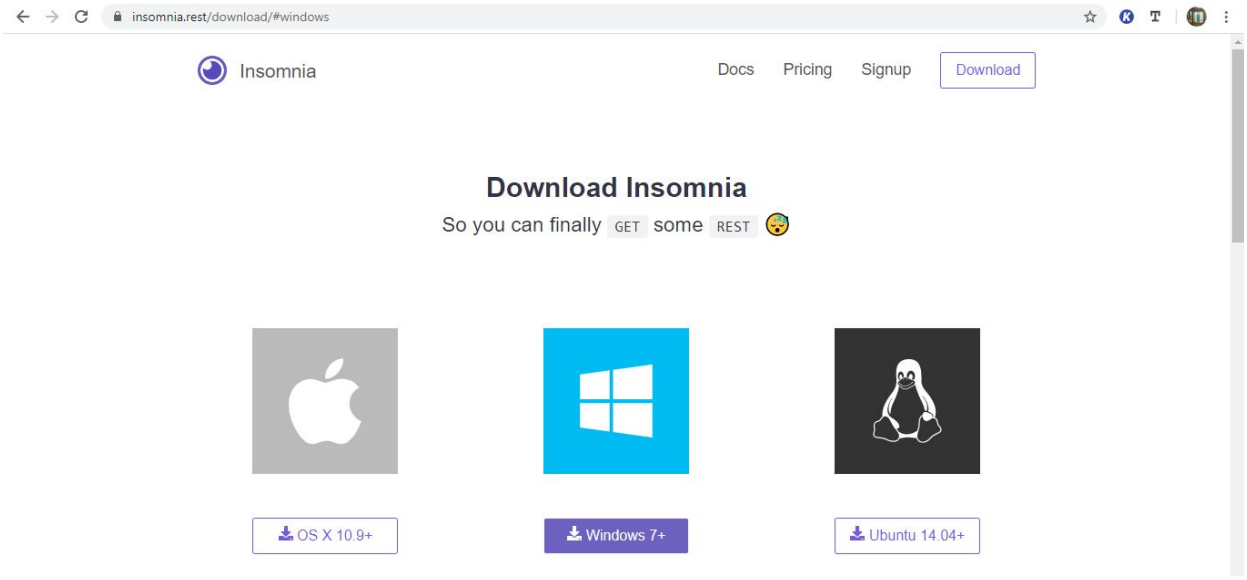
This text is very important, and we will use part of it later.

3.3 Installing Insomnia

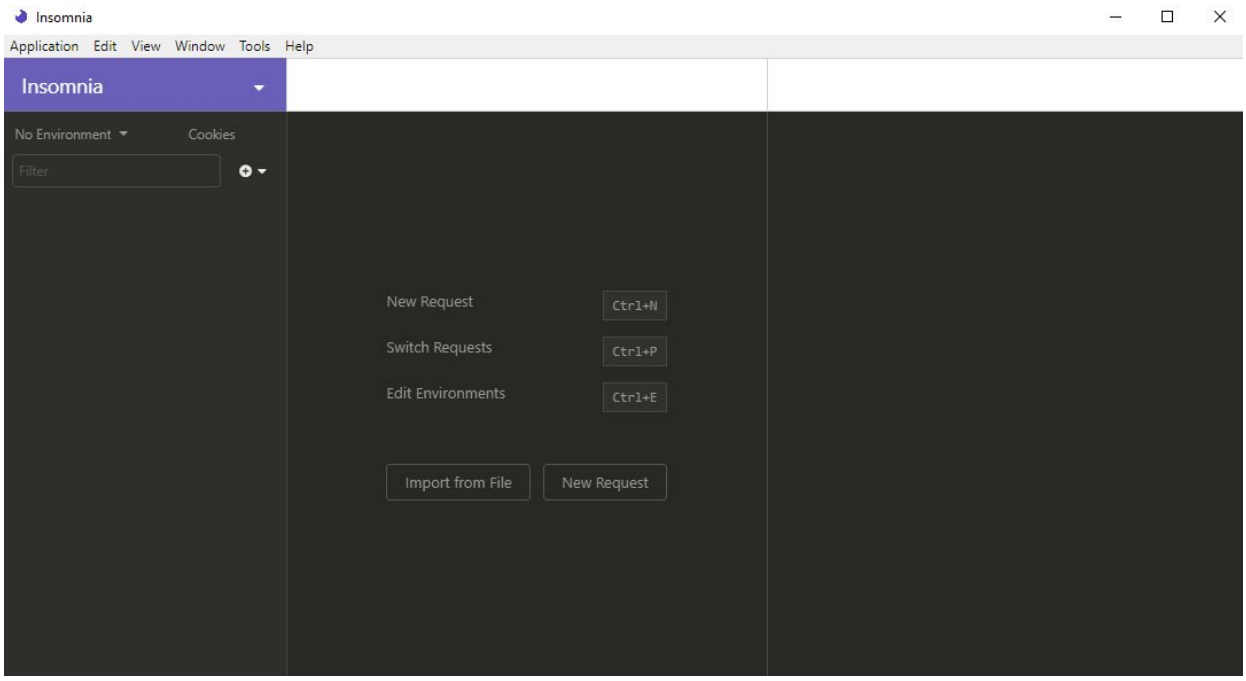
The last part of installation is for you to install Insomnia. This is a tool that will allow us to send commands to any API and get back responses. This is a great tool to use to learn the API of interest. Of course, we will not use such a software in full operation of our code. It is only used as a testing approach to ensure that the queries to APIs that we build are in line with that APIs documentation. To install insomnia, direct your browser to http://insomnia.rest:



Once there, choose your operating system (if you cannot install it on Mac or Ubuntu, I unfortunately cannot help you there. Please try to find a computer on which it will install if you encounter problems). After choosing, a download will begin for the install:



Once downloaded, double click on the file, and a screen that says "Insomnia" will show. It will look like it is frozen. **DO NOT ATTEMPT TO FORCE CLOSE IT**. It is simply installing in the background without a progress bar. If for whatever reason the install process is taking longer than 5 or 10 minutes, then you can attempt to uninstall it and reinstall it. After it installs, you should get to this screen:

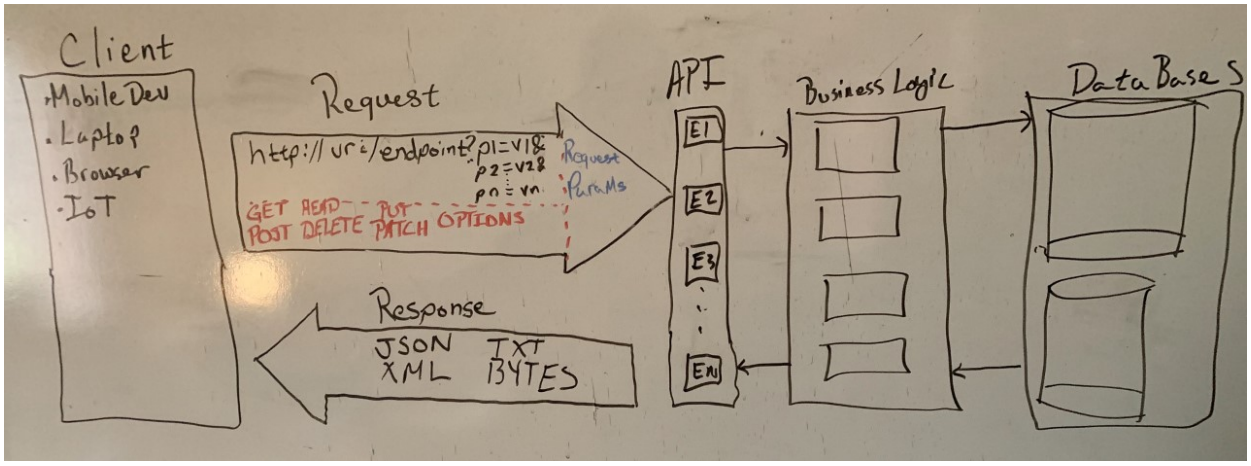


If you do not, look for the software on your computer, and try to load it. Congrats! We are now ready to start the workshop!

4 Tutorial 1: The Basic Structure of an API

As we have mentioned in our intro, an Application Programming Interface (API) is a set of tools that allow us to interface with data, or more appropriate and application, over long distance connections. The API itself acts as a face to a much more intricate application. We can think of an API as an interface to a collection of functions (each of which run not on your computer, but on a different computer), that you can use by passing input over the internet or a network to it and getting an output sent back over to you. Put simply, the API is a front mechanism that dissects requests and processes them. When we say "process", we simply mean that it will take the request to determine:

- Which "library" (i.e. endpoint) (similar to packages in R) should be referenced?
- Which function within the library should be called?
- What are the inputs into the function that should be called?
- What type of output should be returned?



As we can see in the figure above, the API is acting as a liaison between a client and a pre-built server application (i.e. a collection of functions that run on the server). In the diagram above, we notice that we have a few components to explain:

- **The Client**
  The Client is responsible for sending the information. A client can be a web browser, software (as in R or Python), a mobile phone, or a device that is connected to the internet (also known as an Internet of Things (IoT)). The client would like information back by "calling" a function on the server. It uses an API directory to know what input it needs to send and where to send it, as well as what to expect as output (very similar to calling any function in R). It does this by constructing a *request*.

- **The Request**
  A request is group of information that is sent to a server. The server's job is to process the request. The request has a standard format, which usually entails of the following information below (more information about requests can be found at: https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html):

  - Header - Indicates the information about the information that is to be sent. Information such as the method, protocol, and type of information that will be sent is all included in the header.
  - Body - The actual information for the server to process.

  More specifically, a request usually has the following in its header:

  - Method - What should the server do with the message? There are different types:
    * OPTIONS - A request for information regarding how one shall communicate with the server.
    * GET - A request for specific information at a given URI.
    * HEAD - Identical to GET, except the server will not send a body in its response (it will only send a HEADER, which is very useful for certain situations).
    * POST - A request for the server to accept the message body and write or manipulate something on the server (such as a file or a database) with the information given in the body of the message.

    - PUT - Similar to a POST request, but for files, with different meanings on how to handle the provided URI.
    - DELETE - Requests the server to delete information at a given URI.
  - URI - A uniform resource identifier. It indicates the location of the resource upon which the request will be applied. This is usually just a simple URL, but it does not need to be, depending on how the server works.
  - Request-header - Indicates other information that a user would like to send to the server.

- **The API**
  The API acts as the interface between the client and the back-end collection of functions (i.e. the application/business logic). APIs usually comprise of *endpoints*. Each endpoint can be accessed through a unique URL/URI. The easiest way of thinking about an endpoint is similar to how we think about packages in R. That is, endpoints are nothing more than collections of functions to use, grouped together by similarity in their functionality. The API acts as the direct handler of requests coming into the server. When a request comes in, the API looks at the URI and redirects the message (the request) to the appropriate "function" in the "package" (i.e. endpoint). When the function is done and wants to return information back to the client, the API then packages up the information in the form of a *response* (see below for more).

- **The Business Logic**
  The Business Logic is the bread and butter of the API. It is the underlying application. The easiest way to think about the business logic is as nothing more than a single function. Each function serves its own purpose. It has inputs, it does something to the inputs, and provides outputs. This is almost identical to what we discussed in Workshop 1 with writing and using functions in R.

- **The Databases**
  Sometimes, and almost overwhelmingly, business logic will need access to a database. The logic's purpose may be to obtain data from a data base, or, to write new data to a database. Either way, the server and business logic need access to a database. That database can be a more traditional RDMS, which is more structured, or it can be a more modern unstructured database, like NoSQL/Hadoop/Basic File System/etc.

- **The Response**
  Once the business logic is complete doing its thing, it needs to return information (just like any other function does). In R, this is easy to do in a function (we just use the return function). However, on a server, we need to take an additional step, since we are communicating this information over networks. Just how a request is formally structured, so too is a response. When the business logic returns information to the API, the API then takes this information, constructs a response (with a header and a body of the message), and returns it to the client. The body of the response can be anything from binary information, to text, to JSON, to XML, to EDI, to CSV. That is, it can be anything. However, the API will do a good job in writing in the response what type of information is being sent over. The response works by first constructing a response code. The code is usually a three digit number, which follows the following:

  - 1XX - Informational
  - 2XX - Success
  - 3XX - Redirection
  - 4XX - Client Error
  - 5XX - Server Error

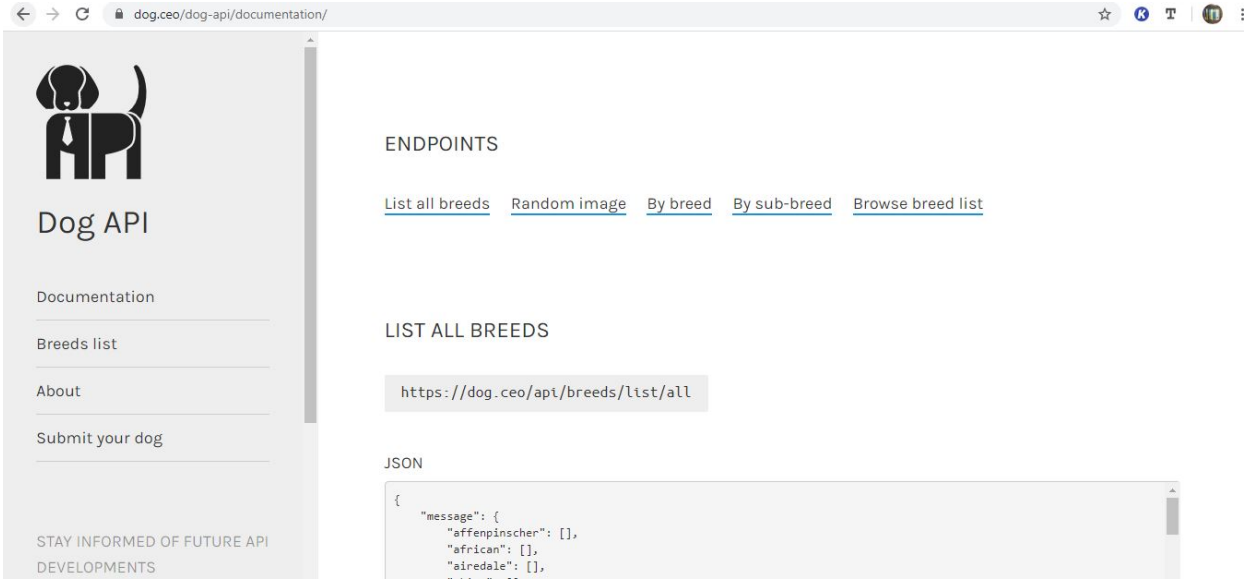
  There are many specific codes. For example, code 202 represents an "Accepted" status, while code 401 represents an "unauthorized" request. The specific request given back from the server helps you indicate any problems you may have had, and how to fix them. Usually, problems arise in responses 4 and 5. If you get a 4XX, then you know its something you did wrong. If you get a 5XX, then it is something that is wrong on the server (at which point, unless you own and operate the server, there is nothing you can do to fix it). Responses will also have the body of the message that it returns. You can find more information on structured responses here: https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html.

So for example, suppose that Twitter has an API (which, it does!). Further suppose you would like to download some tweets for a specific user. Suppose you happen to have that user's unique Twitter screenname. Further suppose that Twitter has an endpoint which is called "statuses", that allows you to download information about tweets. Suppose within this endpoint, there is a function called user_timeline, which allows you to download all of a specific user's Tweets. In order to use this function, which takes a single (and more) required input of the user's screen name, you need to craft a request with (1) the URI to this function and endpoint, (2) the input information, which for this function is the user's screen name. The process by which to accomplish this would be:

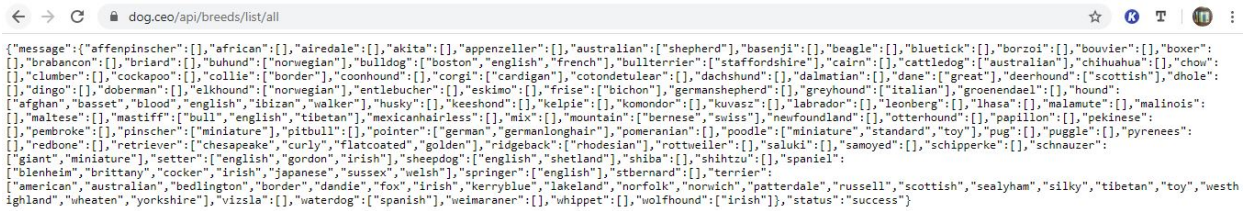
- Get the URI/URL to the specific endpoint (statuses) and function name (user_timelime).
- Get a list of parameter names and values (in this instance, we have only a single input, namely scree_name).
- Determine the request type (usually indicated by the API documentation). For ours, it is a GET request.
- Use the prior three to construct a single request line, which could look like this: `GET https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=twitterapi&count=2`
- Send the request to the server.
- The API reads the request, and directs the input to a function in the business logic (FYI, this function can be written in ANY language such as C, C++, JAVA, R, PYTHON, RUBY, etc.)
- The function in the business logic will go to a database with the provided input of the screen name and look up that users' tweets.
- The database will return this information to the business logic function.
- The business logic function will clean up any information it needs to, or aggregate it if need be, and return it back to the API.

- The API takes the information from the business logic function and packages it up into a response. Depending on what happened, it will assign a specific response code to the response, take the message body (i.e. the information returned by the business logic function), and construct a response string.
- The server sends the response string back to the client, where the client then handles it.
- In the case of Twitter, the tweets will come in the form of JSON. Hence, it is up to the client (i.e. us!) to know what to do with the JSON once we get it back.

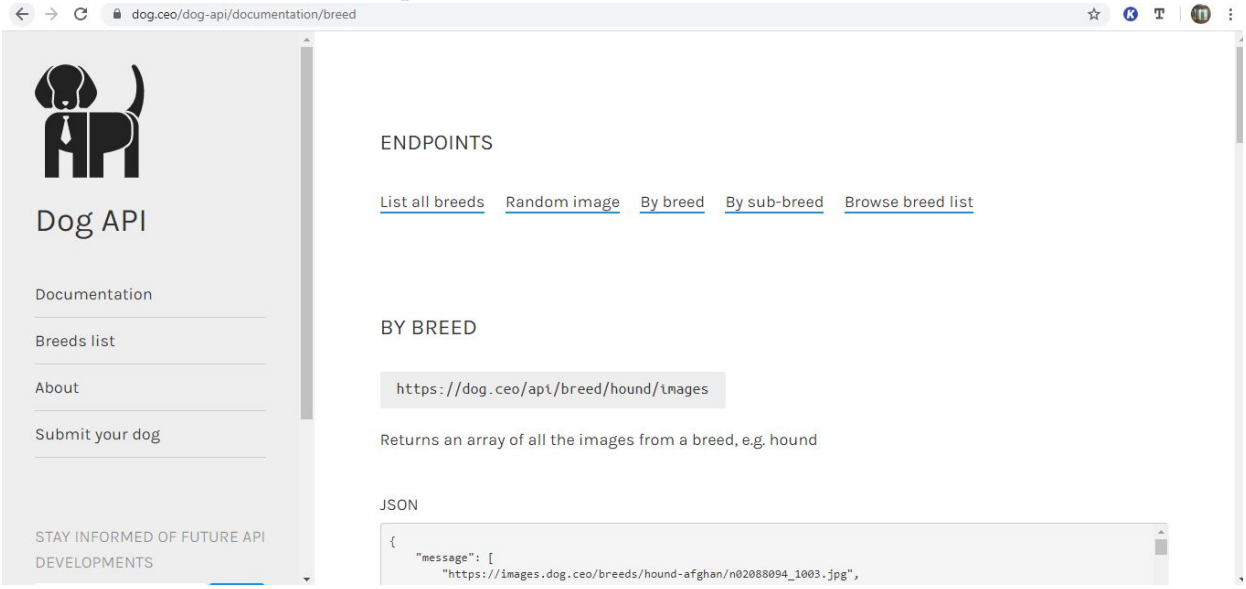
Okay, now to illustrate the use of a real API, we will use a very basic one. The Dog API, found at https://dog.ceo/dog-api/, is an API that allows the user to find and download multiple pictures of different dogs. It is an example API, and it is perfect for our illustration, since it does not necessitate authentication and authorization (more on this later!) First, navigate to the Documentation link on the website:



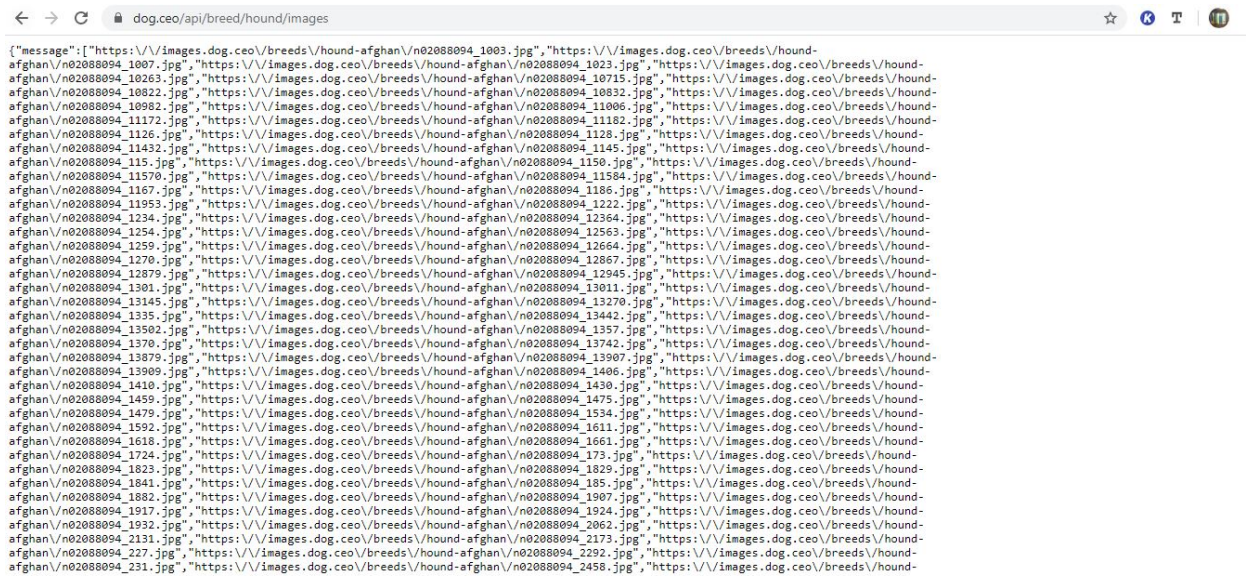
Now, your first API call that you will make is to the endpoint /breeds/list/all. A call to this specific endpoint should return a list, returned in JSON, of all breeds that are in the dog database. Put the link https://dog.ceo/api/breeds/list/all. This is basically a non-input function in R which will return information. Doing this should give you the following:



Now theoretically, you could save this file and further process it. We will not (for the time being at least). But notice the response and the JSON structure. First, it lists the "message" component (which is a list of all breeds of dogs stored in the database). Next notice that it lists the "status" component, which happens to be "success". This means that your request was all good in the eyes of the server. Now let's try a different endpoint. Notice that all of them are listed in the documentation under "Endpoints". Click on the "By breed" endpoint, and the website will give you info on how to access this endpoint:

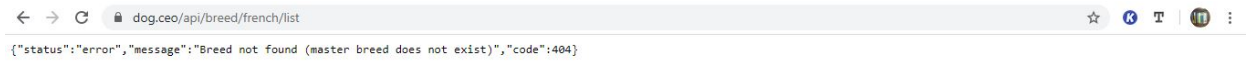


This is actually very quite common with APIs. Usually, an API will have a documentation on how to use it. The documentation will typically be broken into endpoints, where each endpoint will have sub-endpoints (functions), or they will all just simply be endpoints. Each endpoint will have a specific URL/URI, a list of parameters that it can accept, as well as how the response will look. More on this when we get to the next tutorial. Anyway, let us now try to test the breed endpoint. Plug into the browser the endpoint url of https://dog.ceo/api/breed/hound/images. We obtain the following JSON as a result:



Let us try another breed, such as the African breed. We can do this using the url https://dog.ceo/api/breed/african/images

What if we try a "french" breed, if one exists:



So we have gotten a code 404. The server message back to us says that this breed does not exist. Generically, a 404 means: "404" ; Section 10.4.5: Not Found (see https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.5). Put differently, the resource that is requested does not exist at our URL, which means it must be an invalid endpoint.

Exercises

1. Run the following code in your browser:
   https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=twitterapi

   What status code is in the response? Why do you receive this code? (Hint: Look at the response code book in the links provided above)

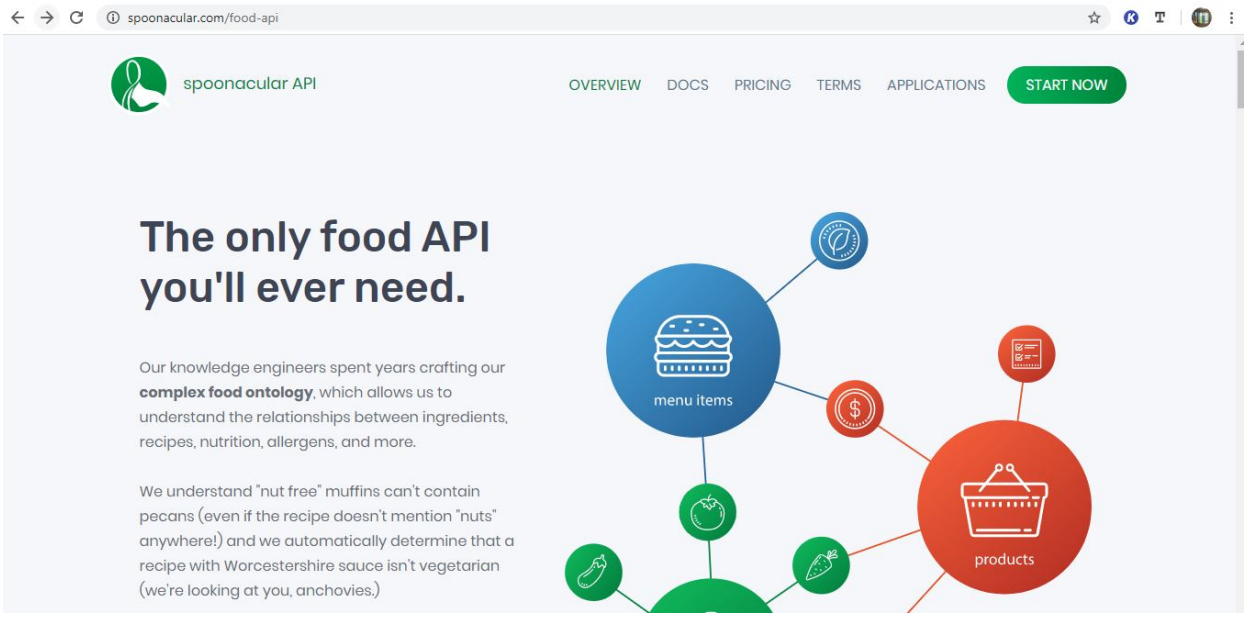
2. In the previous problem, what is the URL for the endpoint? What is the endpoint name? What is the name and value of the parameter passed as input into the function? What do you think the function at this endpoint does?

5 Tutorial 2: How to Learn About APIs

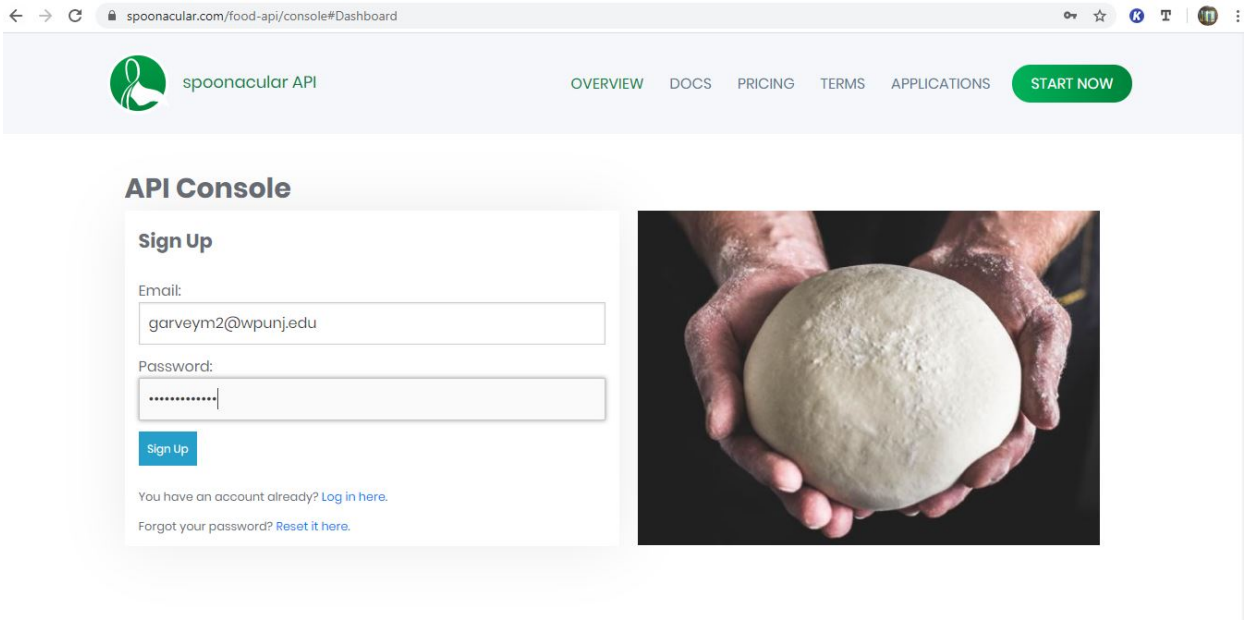
As we saw in the previous tutorial, when we would like to gather data from a website, that website sometimes has an API available to the public to use. Once we know that an API is available to the public, we need to learn how to use it. So, how does one go about learning how to use an API? Recall when you first learned about functions in R that R has documentation that explains what a specific function is used for, what the inputs of that function are, and what the expected outputs of the function are. As it turns out, if an API is made available to the public, it too will most likely have a set of documentation that is similar to that of R.

Let us start simple. Suppose we want to have access to a database full of recipes to use for some type of analysis. Upon Googling around, you will find many of them. A free one that we happen to stumble across is the spoonacular API. While not 100% free, it does allow us to have access to some of its data for free. Usually for a public api, we need to take the following steps to start using one:

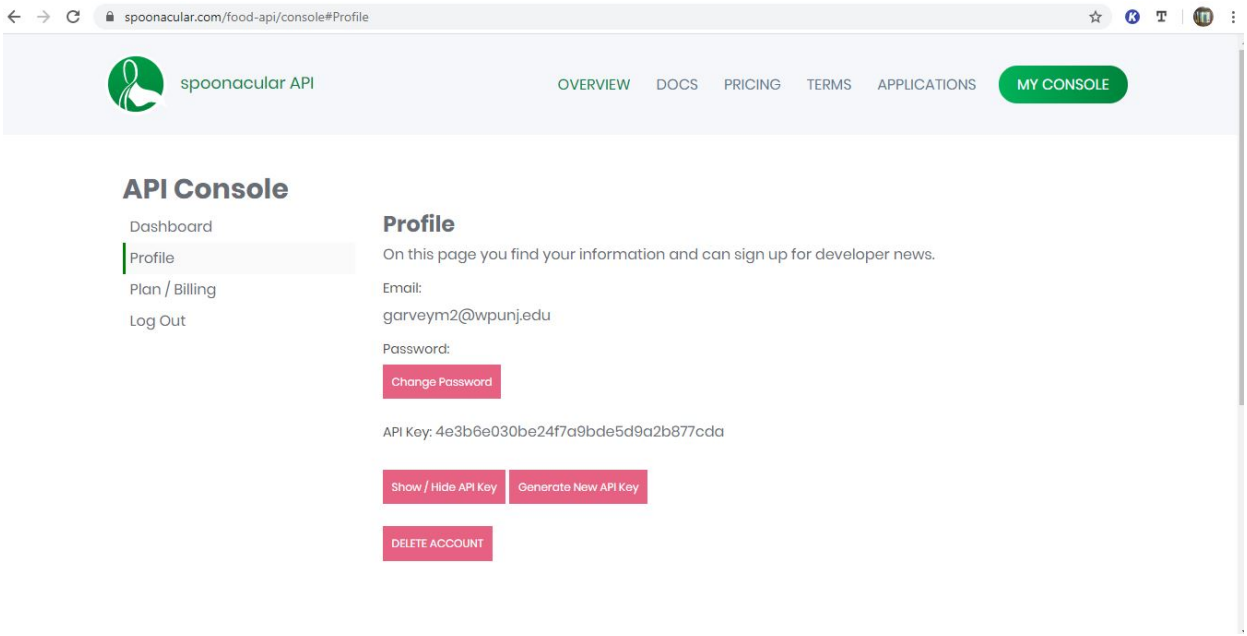
- Sign Up with an Email/Phone
- Verify our account
- Determine the method of authentication to the API. There are many types (Basic Auth/OAuth 1/OAuth2 and more). We absolutely need to learn how to authenticate so that we can actually access the API.
- The Endpoints the API Offers
- The inputs to each endpoint that are required (as well as optional ones).
- The Output Format of the API (Usually it is JSON, but not always).



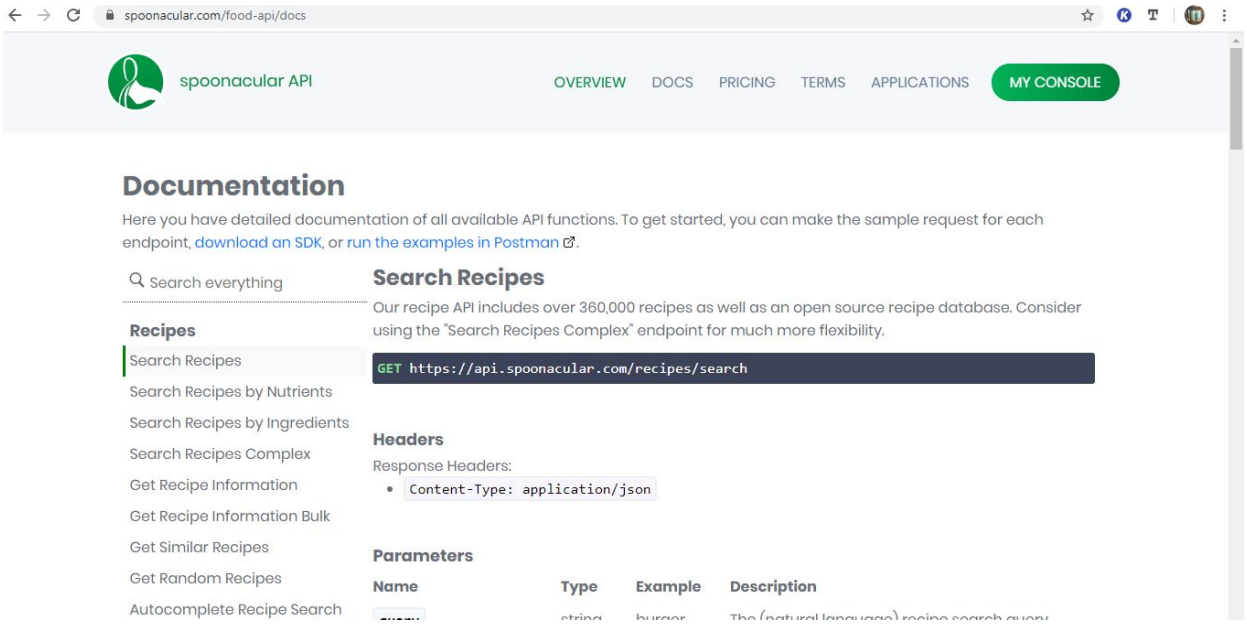
In our case with spoonacular API, let us first sign up. Click the "Start Now" button on the page, and input your email and password to sign up:



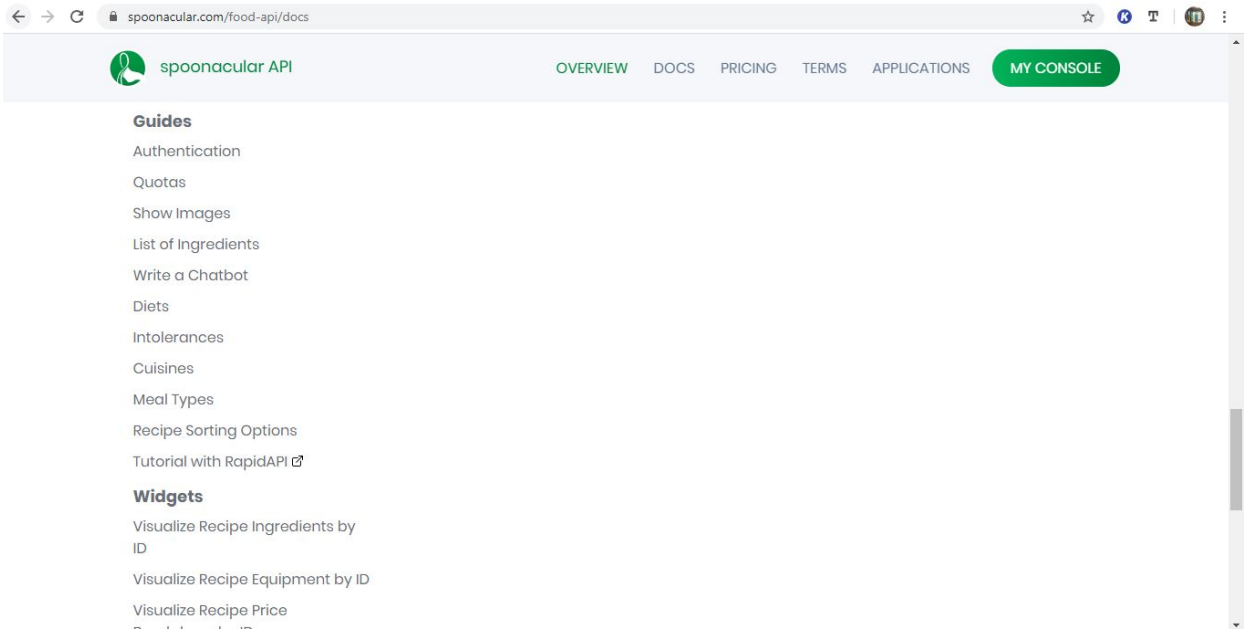
After you sign up, click on "My Console", and then "Profile". You will notice an "API Key". Copy and paste that to a file so you don't loose it:



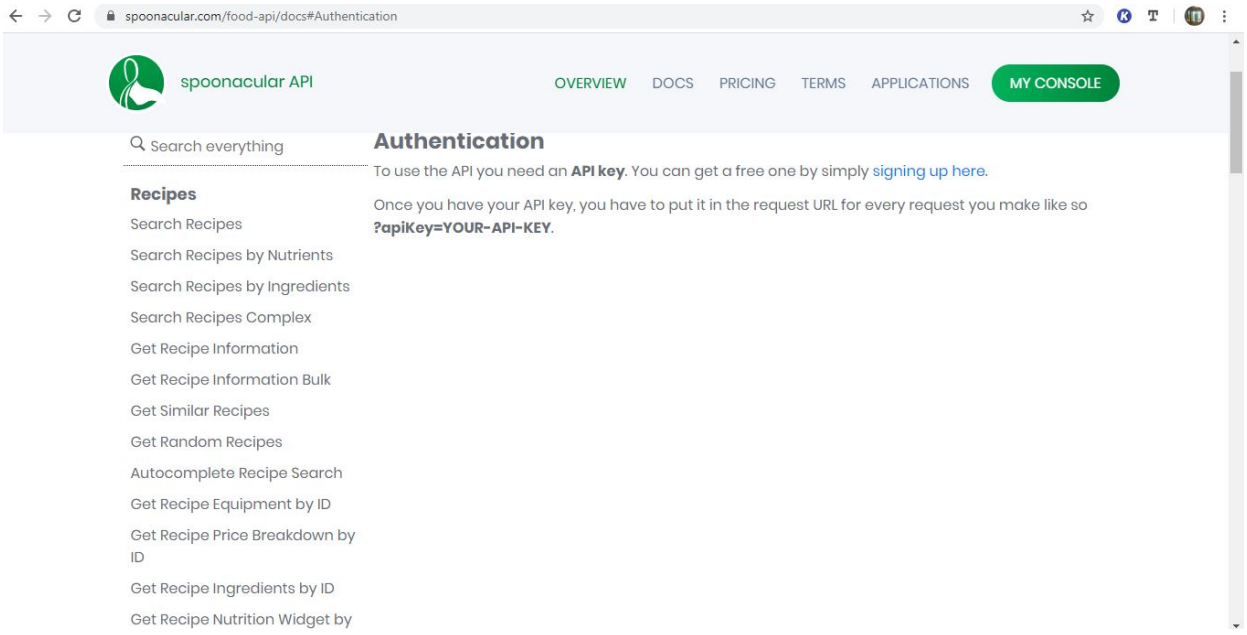
Now looking at our goal here, we need to learn (1) How to authenticate, if it is even necessary to authenticate, (2) the endpoints, (3) inputs for each endpoint, (4) what each endpoint does, (5) the output format for the endpoint. Let us begin. First, let us try to find documentation if it even exists. Luckily, if you look up at the menu bar, you will see "Docs". Common sense would tell us that the API Documentation would need to be there. So let us click it:



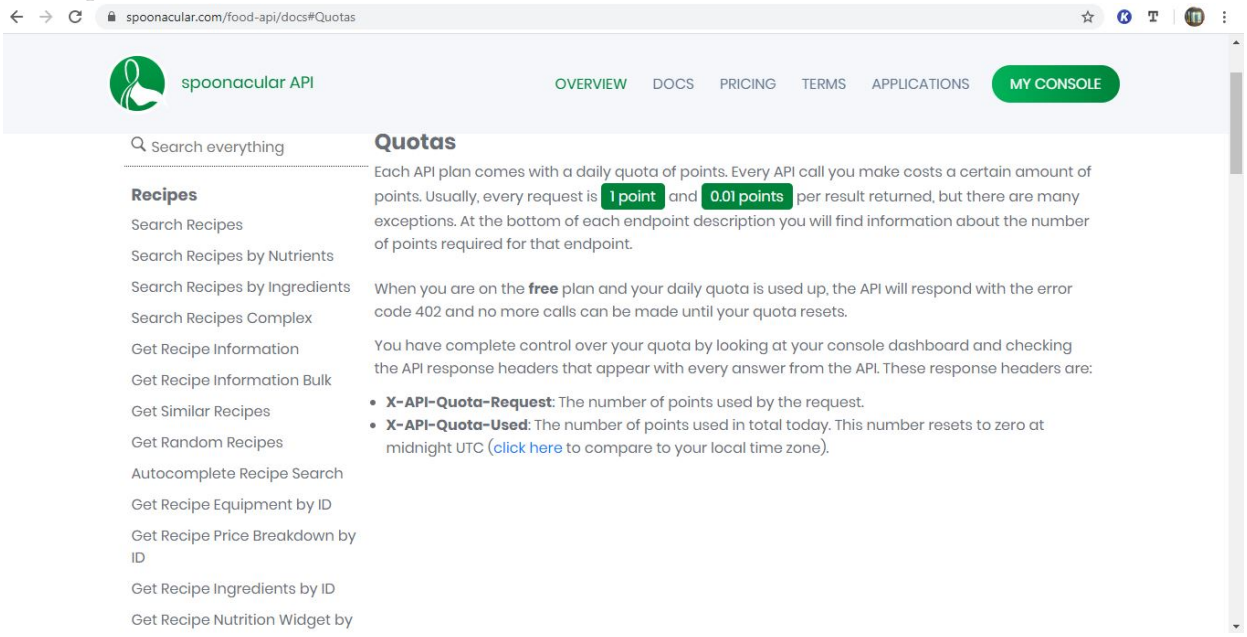
Upon visiting the documentation site, you will notice on the menu on the left side of the screen that there are many links to consider. First, scroll down to see all of them. Specifically, you will notice the screen "Authentication" under the "Guides" menus upon scrolling down:



Clicking on this will reveal how the API authenticates (if it does). Not all API Documentation guides are this clear. Luckily, this one is very clear. In order to authenticate our access to the API, we need to include in our request a parameter named `apiKey` which is assigned to the key given to us on our account (if you didn't record this down, go back to the "My Console" and the "Profile" and copy and paste the key):

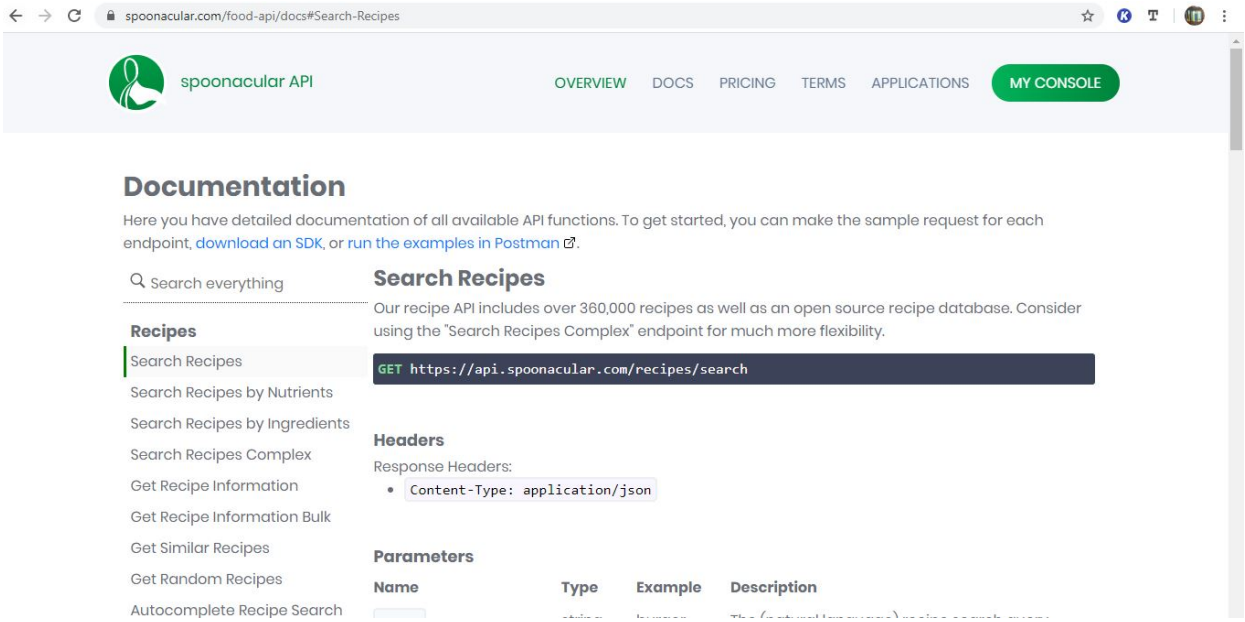


Also notice that there is a "Quotas" guide. While an API will allow you access to a lot of information, it will also restrict your access based on your used. Usually, and API will put a *quota* on the number of requests that you can make in a given amount of time. This API bases its quotas on a point system, which one you surpass the number of points in a given period of time, you will need to wait a while until you can make another request. According to this guide, if you surpassed your quota, the server will respond to your request with a response code of 402 (go look up this code to understand it!):

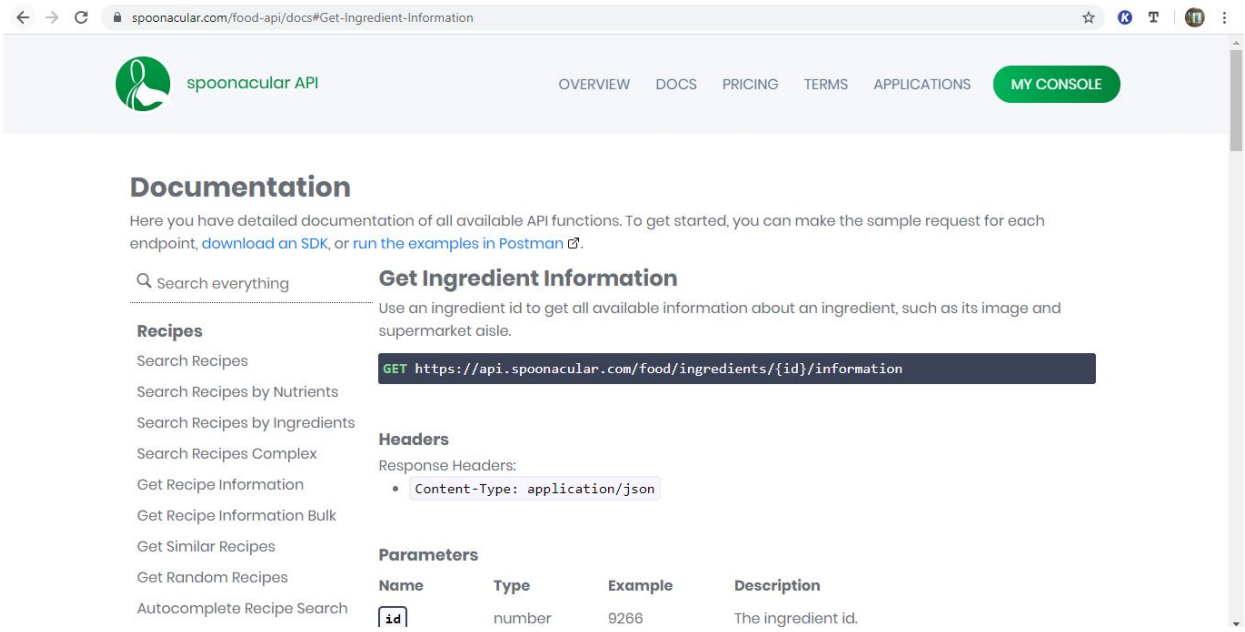


So far, we have learned how to authenticate for this specific API. How about the endpoints? Well, as we can see on the left hand side of the screen, the endpoints appear to have its own documentation page. We notice the endpoints are grouped into seven main endpoints, with each group having their own endpoints: Recipes, Ingredients, Products, Menu Items, Meal Planning, Wine, and Miscellaneous. Think of these main groups as you would think of "packages" in R,

with each endpoint under each category as its own "function" in R. The question is now, how do we use these? First, click on "Search Recipes" on the left under the "Recipes" section, which will bring you to here:



On this page, we notice a few things. First, the documentation indicates what this endpoint (or, "function) does in the first place. The second piece of information it provides is the request string that is needed in order to make a call to this endpoint. Usually this includes the URL at which the endpoint is located. The third piece of information is any header information that the endpoint will return in its response. The fourth piece of information it provides is the list of parameters. Think of this list as the "inputs" into the function "search". Each input is indicated by its name, the type of information that is to be passed as an input, an example of input that can be passed, and a description of what is the intent of the input. It would appear that bold faced inputs are required, while non-bold faced are optional inputs. The fifth piece of information it provides is an example request as well as response. The examples further help the user learn how to use the API by understanding how to structure their request as well as what output can be expected. Notice that we get the same type of information if we click on "Get Ingredient Information" below:



The guide therefore will help you better understand how to use the specific API. For example, suppose we wanted to search for recipes based on the string "Burger" but excluding any recipes that use beef. We go to the documentation that can search for recipes, as we had done before. We know that in order to make a call to the API, we need to construct a request. The guide for "Search Recipes" tells us that it is a GET request. Furthermore, the guide told us that authentication occurs by providing the parameter "apiKey" in the request string. The guide tells us that we can search for a recipe by providing a value to the parameter "query". Likewise, reading the documentation, we notice that we can exclude ingredients by using the "excludeIngredients" parameter. Suppose we only want the API to give us 5 recipes. We can use the "number" parameter to indicate this. Therefore, we have the following information to construct our request based on our needs and the documentation:

- Endpoint URL: https://api.spoonacular.com/recipes/search
- Parameters and Values:
  - apiKey = 4e3b6e030be24f7a9bde5d9a2b877cda (You need to input your own from your own account)
  - query = Burger
  - excludeIngredients = beef
  - number = 5
- Response: JSON

Using this information, we can build the request url string. We do this by first specifying the URL to the endpoint, followed by a "?". The "?" tells the server that the text to the right of "?" is a list of parameters and values. Each parameter = value combination is separated in the string by an "&". Hence, we have:

```
https://api.spoonacular.com/recipes/search?apiKey=4e3b6e030be24f7a9bde5d9a2b877cda
    &query=Burger&excludeIngredients=beef&number=5
```

Listing 2: Constructing the Request URL.

If you copy and paste this into a web browser, you will obtain the following output in JSON:


```
{"results":[{"id":246916,"title":"Bison Burger","readyInMinutes":45,"servings":6,"
    image":"Buffalo-Burger-246916.jpg","imageUrls":["Buffalo-Burger-246916.jpg"]},{
    "id":245166,"title":"Hawaiian Pork Burger","readyInMinutes":40,"servings":4,"
    image":"Hawaiian-Pork-Burger-245166.jpg","imageUrls":["Hawaiian-Pork-Burger
    -245166.jpg"]},{"id":219957,"title":"Carrot & sesame burgers","readyInMinutes"
    :50,"servings":6,"image":"Carrot---sesame-burgers-219957.jpg","imageUrls":["
    Carrot---sesame-burgers-219957.jpg"]},{"id":607109,"title":"Turkey Zucchini
    Burger with Garlic Mayo","readyInMinutes":45,"servings":6,"image":"Turkey-
    Zucchini-Burger-with-Garlic-Mayo-607109.jpg","imageUrls":["Turkey-Zucchini-
    Burger-with-Garlic-Mayo-607109.jpg"]},{"id":219871,"title":"Halloumi aubergine
    burgers with harissa relish","readyInMinutes":20,"servings":4,"image":"Halloumi
    -aubergine-burgers-with-harissa-relish-219871.jpg","imageUrls":["Halloumi-
    aubergine-burgers-with-harissa-relish-219871.jpg"]}],"baseUri":"https://
    spoonacular.com/recipeImages/","offset":0,"number":5,"totalResults":67,"
    processingTimeMs":111,"expires":1578751620313}
```


Listing 3: Output JSON Unformatted.

This is obviously messy and hard to read (as we had seen before). You may want to have an online software automatically format it so it is a bit easier on your eyes. One such website is https://jsonformatter.curiousconcept.com/. After formatting, we obtain:


```
{
    "results":[
    {
        "id":246916,
        "title":"Bison Burger",
        "readyInMinutes":45,
        "servings":6,
        "image":"Buffalo-Burger-246916.jpg",
        "imageUrls":[
            "Buffalo-Burger-246916.jpg"
        ]
    },
    {
        "id":245166,
        "title":"Hawaiian Pork Burger",
        "readyInMinutes":40,
        "servings":4,
        "image":"Hawaiian-Pork-Burger-245166.jpg",
        "imageUrls":[
            "Hawaiian-Pork-Burger-245166.jpg"
        ]
    },
    {
        "id":219957,
        "title":"Carrot & sesame burgers",
        "readyInMinutes":50,
        "servings":6,
        "image":"Carrot---sesame-burgers-219957.jpg",
        "imageUrls":[
            "Carrot---sesame-burgers-219957.jpg"
        ]
    },
    {
```

```
        "id":607109,
        "title":"Turkey Zucchini Burger with Garlic Mayo",
        "readyInMinutes":45,
        "servings":6,
        "image":"Turkey-Zucchini-Burger-with-Garlic-Mayo-607109.jpg",
        "imageUrls":[
            "Turkey-Zucchini-Burger-with-Garlic-Mayo-607109.jpg"
        ]
        },
    {
        "id":219871,
        "title":"Halloumi aubergine burgers with harissa relish",
        "readyInMinutes":20,
        "servings":4,
        "image":"Halloumi-aubergine-burgers-with-harissa-relish-219871.jpg",
        "imageUrls":[
            "Halloumi-aubergine-burgers-with-harissa-relish-219871.jpg"
        ]
    }
    ],
    "baseUri":"https://spoonacular.com/recipeImages/",
    "offset":0,
    "number":5,
    "totalResults":67,
    "processingTimeMs":111,
    "expires":1578751620313
}
```


Listing 4: JSON Output Formatted.

Notice now that the response structure is fairly straightforward. We have 7 parameters returned to us:

- results - a list of recipes matching our query.
- baseUri - information about the recipe images.
- offset - the initial value set to offset in the original request.
- number - the number of results returned.
- totalResults - the total number of results found in the API Database matching our query.
- processingTimeMs - the total number of milliseconds our request took to process.
- expires - the timestamp in ms that our request will expire.

What we care the most about, however, is the list of results. We can see that a single result has the following structure:

- id - the unique id that specifies the recipe
- title - the name of the recipe
- readyInMinutes - the amount of time it takes to cook the recipe
- servings - the number of servings the recipe will yield

- image - the url to an image of the recipe
- imageUrls - a list of urls to more images of the recipe.

Suppose we wanted the actual recipe of the first dish. We can again go to the API Documentation to figure out how to do this. Notice that if you click on "Get Recipe Information", it seems that this is an endpoint that can give us the information we need. Going through a similar process as before, we have the following information needed to construct the request string:

- Endpoint URL: `https://api.spoonacular.com/recipes/{id}/information`
- Replace {id} with 246916, since this is the dish we want to get information for.
- Parameters and Values:
  - apiKey = 4e3b6e030be24f7a9bde5d9a2b877cda (You need to input your own from your own account)
- Response: JSON

```
https://api.spoonacular.com/recipes/246916/information?apiKey=4
    e3b6e030be24f7a9bde5d9a2b877cda
```

Listing 5: Constructing the Request URL.

After running this in your web browser, you will get more JSON output. Formatting it through the online formatter, we can see the following:


```
{
   "vegetarian":false,
   "vegan":false,
   "glutenFree":true,
   "dairyFree":true,
   "veryHealthy":false,
   "cheap":false,
   "veryPopular":true,
   "sustainable":false,
   "weightWatcherSmartPoints":12,
   "gaps":"no",
   "lowFodmap":false,
   "ketogenic":false,
   "whole30":false,
   "sourceUrl":"https://www.simplyrecipes.com/recipes/buffalo_burger/",
   "spoonacularSourceUrl":"https://spoonacular.com/bison-burger-246916",
   "aggregateLikes":5345,
   "spoonacularScore":93.0,
   "healthScore":24.0,
   "creditsText":"Simply Recipes",
   "sourceName":"Simply Recipes",
   "pricePerServing":349.0,
   "extendedIngredients":[
      {
         "id":17330,
```

```
        "aisle":"Meat",
        "image":"fresh-ground-beef.jpg",
        "consitency":"solid",
        "name":"ground bison",
        "original":"2 pounds ground bison (buffalo)",
        "originalString":"2 pounds ground bison (buffalo)",
        "originalName":"ground bison (buffalo)",
        "amount":2.0,
        "unit":"pounds",
        "meta":[
           "(buffalo)"
        ],
        "metaInformation":[
           "(buffalo)"
        ],
        "measures":{
           "us":{
              "amount":2.0,
              "unitShort":"lb",
              "unitLong":"pounds"
           },
           "metric":{
              "amount":907.185,
              "unitShort":"g",
              "unitLong":"grams"
           }
        }
     },
     {
        "id":99226,
        "aisle":"Produce;Spices and Seasonings",
        "image":"fresh-sage.png",
        "consitency":"solid",
        "name":"fresh sage",
        "original":"2 tablespoons finely chopped fresh sage",
        "originalString":"2 tablespoons finely chopped fresh sage",
        "originalName":"finely chopped fresh sage",
        "amount":2.0,
        "unit":"tablespoons",
        "meta":[
           "fresh",
           "finely chopped"
        ],
        "metaInformation":[
           "fresh",
           "finely chopped"
        ],
        "measures":{
           "us":{
              "amount":2.0,
              "unitShort":"Tbsps",
              "unitLong":"Tbsps"
           },
           "metric":{
              "amount":2.0,
              "unitShort":"Tbsps",
```

```
                "unitLong":"Tbsps"
            }
        }
    },
    {
        "id":2047,
        "aisle":"Spices and Seasonings",
        "image":"salt.jpg",
        "consitency":"solid",
        "name":"salt",
        "original":"1 1/2 teaspoons salt",
        "originalString":"1 1/2 teaspoons salt",
        "originalName":"salt",
        "amount":1.5,
        "unit":"teaspoons",
        "meta":[

        ],
        "metaInformation":[

        ],
        "measures":{
            "us":{
                "amount":1.5,
                "unitShort":"tsps",
                "unitLong":"teaspoons"
            },
            "metric":{
                "amount":1.5,
                "unitShort":"tsps",
                "unitLong":"teaspoons"
            }
        }
    },
    {
        "id":1002030,
        "aisle":"Spices and Seasonings",
        "image":"pepper.jpg",
        "consitency":"solid",
        "name":"black pepper",
        "original":"2 teaspoons black pepper",
        "originalString":"2 teaspoons black pepper",
        "originalName":"black pepper",
        "amount":2.0,
        "unit":"teaspoons",
        "meta":[
            "black"
        ],
        "metaInformation":[
            "black"
        ],
        "measures":{
            "us":{
                "amount":2.0,
                "unitShort":"tsps",
                "unitLong":"teaspoons"
```

```
            },
            "metric":{
               "amount":2.0,
               "unitShort":"tsps",
               "unitLong":"teaspoons"
            }
         }
      },
      {
         "id":11282,
         "aisle":"Produce",
         "image":"brown-onion.png",
         "consitency":"solid",
         "name":"onion",
         "original":"1/2 onion, finely chopped",
         "originalString":"1/2 onion, finely chopped",
         "originalName":"onion, finely chopped",
         "amount":0.5,
         "unit":"",
         "meta":[
            "finely chopped"
         ],
         "metaInformation":[
            "finely chopped"
         ],
         "measures":{
            "us":{
               "amount":0.5,
               "unitShort":"",
               "unitLong":""
            },
            "metric":{
               "amount":0.5,
               "unitShort":"",
               "unitLong":""
            }
         }
      },
      {
         "id":1034053,
         "aisle":"Oil, Vinegar, Salad Dressing",
         "image":"olive-oil.jpg",
         "consitency":"liquid",
         "name":"extra virgin olive oil",
         "original":"2 tablespoons extra virgin olive oil",
         "originalString":"2 tablespoons extra virgin olive oil",
         "originalName":"extra virgin olive oil",
         "amount":2.0,
         "unit":"tablespoons",
         "meta":[

         ],
         "metaInformation":[

         ],
         "measures":{
```

```
            "us":{
               "amount":2.0,
               "unitShort":"Tbsps",
               "unitLong":"Tbsps"
            },
            "metric":{
               "amount":2.0,
               "unitShort":"Tbsps",
               "unitLong":"Tbsps"
            }
         }
      },
      {
         "id":6150,
         "aisle":"Condiments",
         "image":"barbecue-sauce.jpg",
         "consitency":"solid",
         "name":"barbecue sauce",
         "original":"Smoky barbecue sauce",
         "originalString":"Smoky barbecue sauce",
         "originalName":"Smoky barbecue sauce",
         "amount":1.0,
         "unit":"serving",
         "meta":[

         ],
         "metaInformation":[

         ],
         "measures":{
            "us":{
               "amount":1.0,
               "unitShort":"serving",
               "unitLong":"serving"
            },
            "metric":{
               "amount":1.0,
               "unitShort":"serving",
               "unitLong":"serving"
            }
         }
      }
   ],
   "id":246916,
   "title":"Bison Burger",
   "readyInMinutes":45,
   "servings":6,
   "image":"https://spoonacular.com/recipeImages/246916-556x370.jpg",
   "imageType":"jpg",
   "cuisines":[
      "American"
   ],
   "dishTypes":[
      "lunch",
      "main course",
      "main dish",
```

```
      "dinner"
   ],
   "diets":[
      "gluten free",
      "dairy free"
   ],
   "occasions":[

   ],
   "winePairing":{
      "pairedWines":[
         "malbec",
         "merlot",
         "zinfandel"
      ],
      "pairingText":"Malbec, Merlot, and Zinfandel are great choices for Bison Burger. Merlot will be perfectly adequate for a classic burger with standard toppings. Bolder toppings call for bolder wines, such as a malbec or peppery zinfandel. You could try Terrazas de los Andes Reserva Malbec. Reviewers quite like it with a 4.2 out of 5 star rating and a price of about 22 dollars per bottle.",
      "productMatches":[
         {
            "id":438817,
            "title":"Terrazas de los Andes Reserva Malbec",
            "description":"Bright red color with purple shades. Intense floral and fruity notes. Presence of violets, ripe black cherry and plum aromas. ReIts sweet and juicy mouthfeel delivers finesse, delicate tannins and an elegant finish of black fruits. veals a toasty and spicy character of black pepper and chocolate.Pair with Tuna Carpaccio and Red Fruits Vinaigrette, Tenderloin served with MediterraneanRatatouille and Moist Semisweet Chocolate Cake over Mascarpone and Red Fruits Ice Cream.",
            "price":"$21.99",
            "imageUrl":"https://spoonacular.com/productImages/438817-312x231.jpg",
            "averageRating":0.8400000000000001,
            "ratingCount":5.0,
            "score":0.7775000000000001,
            "link":"https://click.linksynergy.com/deeplink?id=*QCiIS6t4gA&mid=2025&murl=https%3A%2F%2Fwww.wine.com%2Fproduct%2Fterrazas-de-los-andes-reserva-malbec-2014%2F162923"
         }
      ]
   },
   "instructions":"Cook the onions: Saut  the onions in the olive oil over medium-high heat until translucent. Turn off the heat and let it cool.\n\nMix onions, sage, salt, pepper into ground bison meat: When the onions are cool enough to touch, use your (clean) hands to gently mix them in with the bison burger meat, and add everything else.\nDo not overwork the meat, it will result in a tough burger. Just gently fold it until the onions, sage, salt and pepper are well mixed in.\n\nForm patties: Form patties with the meat, using about 1/4 to 1/3 of a pound of meat per patty.\nHere's a tip on making the patty: if you press a slight indentation in the center of each patty it will help keep the burgers in a nice disk shape when cooking. Otherwise the burger will start to get a little egg-shaped as the edges contract from cooking.\n\nGrill or fry the burgers: Grill or fry the burgers on medium heat, about 6-7 minutes per side, less or more depending on the thickness of the burger and the heat of the pan/
```

```
grill, or until the internal temperature is 140 F  for medium rare, or 160 F for well done.\nA note on internal temperature. If you are getting the ground meat from a source you trust (we got ours from Whole Foods) that does their own grinding on site, or you grind your own meat, you can safely cook the burgers rare or medium rare. Otherwise you'll want to cook the burgers until well done.\nDont press on your burgers while cooking, and keep the flipping to a minimum.\nLet the burgers rest about 5 minutes before serving.\nServe the burger with lettuce and tomato, topped with a smoky barbecue sauce.",
"analyzedInstructions":[
   {
      "name":"",
      "steps":[
         {
            "number":1,
            "step":"Cook the onions: Saut  the onions in the olive oil over medium-high heat until translucent. Turn off the heat and let it cool.",
            "ingredients":[
               {
                  "id":4053,
                  "name":"olive oil",
                  "image":"olive-oil.jpg"
               },
               {
                  "id":11282,
                  "name":"onion",
                  "image":"brown-onion.png"
               }
            ],
            "equipment":[

            ]
         },
         {
            "number":2,
            "step":"Mix onions, sage, salt, pepper into ground bison meat: When the onions are cool enough to touch, use your (clean) hands to gently mix them in with the bison burger meat, and add everything else.",
            "ingredients":[
               {
                  "id":17330,
                  "name":"ground bison",
                  "image":"fresh-ground-beef.jpg"
               },
               {
                  "id":11282,
                  "name":"onion",
                  "image":"brown-onion.png"
               },
               {
                  "id":1002030,
                  "name":"pepper",
                  "image":"pepper.jpg"
               },
               {
                  "id":17330,
                  "name":"bison",
```

```
                    "image":"fresh-ground-beef.jpg"
                },
                {
                    "id":99226,
                    "name":"sage",
                    "image":"fresh-sage.png"
                },
                {
                    "id":2047,
                    "name":"salt",
                    "image":"salt.jpg"
                }
            ],
            "equipment":[

            ]
        },
        {
            "number":3,
            "step":"Do not overwork the meat, it will result in a tough burger. Just gently fold it until the onions, sage, salt and pepper are well mixed in.",
            "ingredients":[
                {
                    "id":1102047,
                    "name":"salt and pepper",
                    "image":"salt-and-pepper.jpg"
                },
                {
                    "id":11282,
                    "name":"onion",
                    "image":"brown-onion.png"
                },
                {
                    "id":99226,
                    "name":"sage",
                    "image":"fresh-sage.png"
                }
            ],
            "equipment":[

            ]
        },
        {
            "number":4,
            "step":"Form patties: Form patties with the meat, using about 1/4 to 1/3 of a pound of meat per patty.",
            "ingredients":[

            ],
            "equipment":[

            ]
        },
        {
            "number":5,
```

```
            "step":"Here's a tip on making the patty: if you press a slight indentation in the center of each patty it will help keep the burgers in a nice disk shape when cooking. Otherwise the burger will start to get a little egg-shaped as the edges contract from cooking.",
            "ingredients":[

            ],
            "equipment":[

            ]
         },
         {
            "number":6,
            "step":"Grill or fry the burgers: Grill or fry the burgers on medium heat, about 6-7 minutes per side, less or more depending on the thickness of the burger and the heat of the pan/grill, or until the internal temperature is 140 F  for medium rare, or 160 F  for well done.",
            "ingredients":[

            ],
            "equipment":[
               {
                  "id":404706,
                  "name":"grill",
                  "image":"grill.jpg"
               },
               {
                  "id":404645,
                  "name":"frying pan",
                  "image":"pan.png"
               }
            ],
            "length":{
               "number":7,
               "unit":"minutes"
            }
         },
         {
            "number":7,
            "step":"A note on internal temperature. If you are getting the ground meat from a source you trust (we got ours from Whole Foods) that does their own grinding on site, or you grind your own meat, you can safely cook the burgers rare or medium rare. Otherwise you'll want to cook the burgers until well done.",
            "ingredients":[

            ],
            "equipment":[

            ]
         },
         {
            "number":8,
            "step":" D o n t  press on your burgers while cooking, and keep the flipping to a minimum.",
            "ingredients":[
```

```
                ],
                "equipment":[

                ]
            },
            {
                "number":9,
                "step":"Let the burgers rest about 5 minutes before serving.",
                "ingredients":[

                ],
                "equipment":[

                ],
                "length":{
                    "number":5,
                    "unit":"minutes"
                }
            },
            {
                "number":10,
                "step":"Serve the burger with lettuce and tomato, topped with a smoky barbecue sauce.",
                "ingredients":[
                    {
                        "id":6150,
                        "name":"barbecue sauce",
                        "image":"barbecue-sauce.jpg"
                    }
                ],
                "equipment":[

                ]
            }
        ]
    }
]
}
```


Listing 6: Formatted JSON Output.

Needless to say, this is a lot of information for a single recipe. Of course, we would need to dissect this information further to understand how to process, potentially analyze, and/or store this information. For now, we will not do so. As you can see, however, learning APIs are fairly straightforward. Again, it boils down to (1) how does authentication work, (2) what are the endpoints, (3) what does each endpoint do, (4) what are the required inputs (parameters) and optional inputs to the endpoint, and (5) what is the anticipated output format and structure? The API Documentation will help guide you to answer these questions.

Exercises

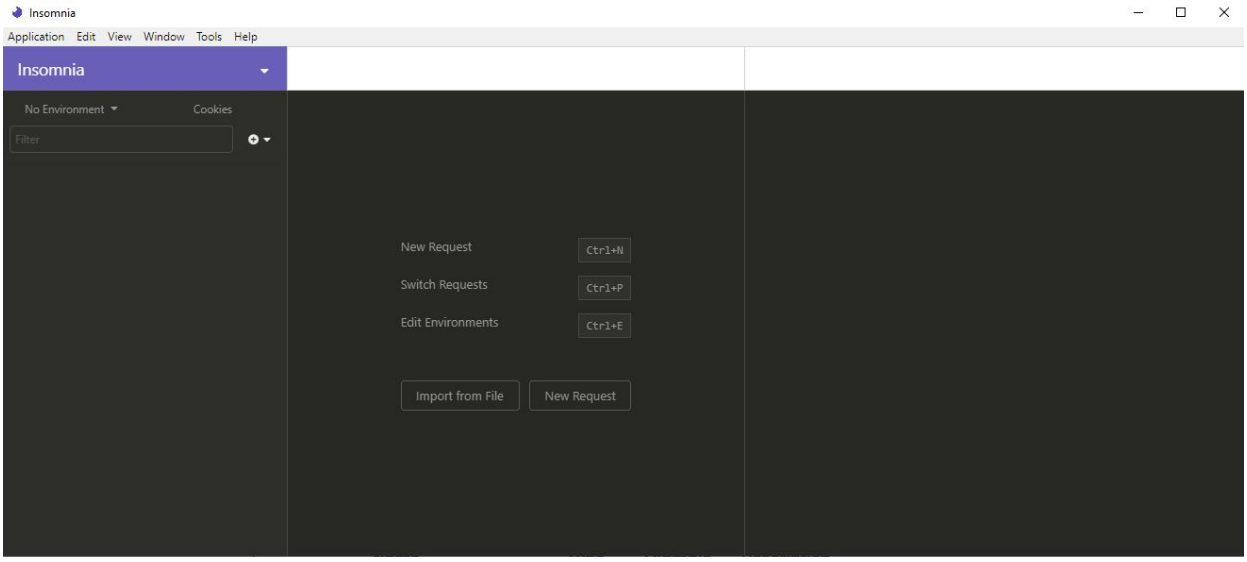
3. Go to http://newsapi.org. Look up the documentation. How does one authenticate themselves with this particular API? What do they need to do?

4. Find one endpoint on this API. Explain what it does, what the URL for it is, and what the parameters do. Also specify which parameters are necessary and which ones are optional.

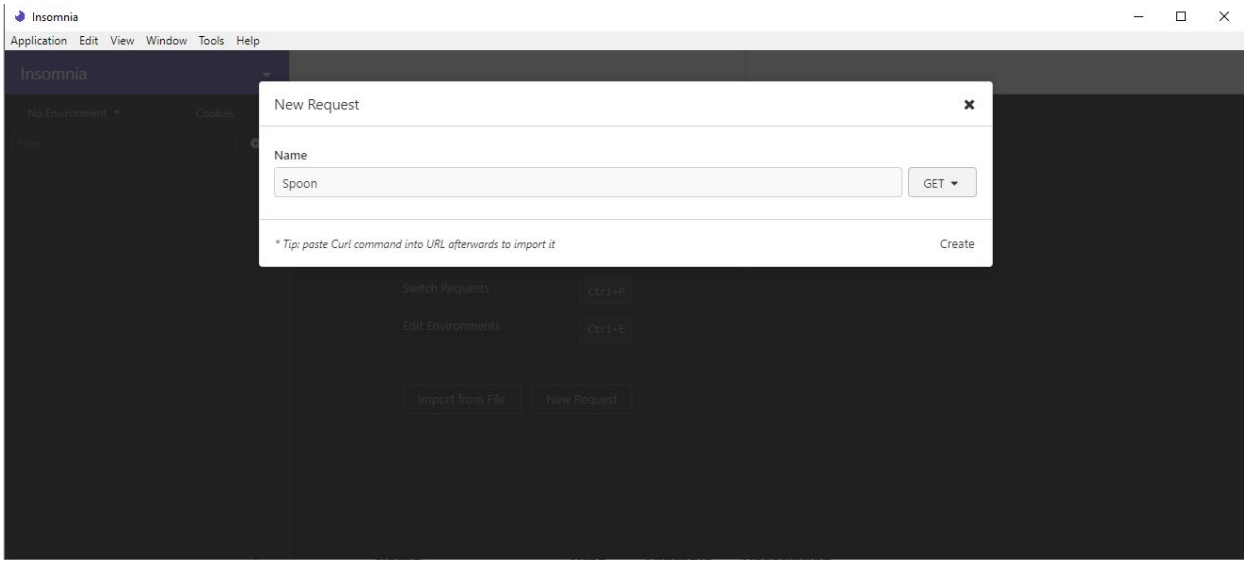
6 Tutorial 3: Gathering Data with Yellow Pages and Twitter in R

We now can finally move on to the final portion of our workshop, which is accessing an API from outside of a web browser, and more specifically, in R. Lucky for us, R has some nice packages that allow us to write requests for APIs. Some packages, as we will see soon, have even been designed so that we do not need to process JSON. Before we get to R, let us understand how other authentication methods work. Up to this point, we have been constructing our requests by simply designing a URL, with the main hyperlink, followed by a list of parameters and values. We primarily "authenticated" ourselves by adding in a token directly as a parameter. However, most APIs do not do this. Most use a mechanism that is called OAuth 1 or OAuth 2. Typically, this process does not allow us to make requests via browsers. Instead, we need special software (in our case, R and Insomnia), which will take our token information, along with our parameter list and endpoint url, and it will construct a request for us automatically.

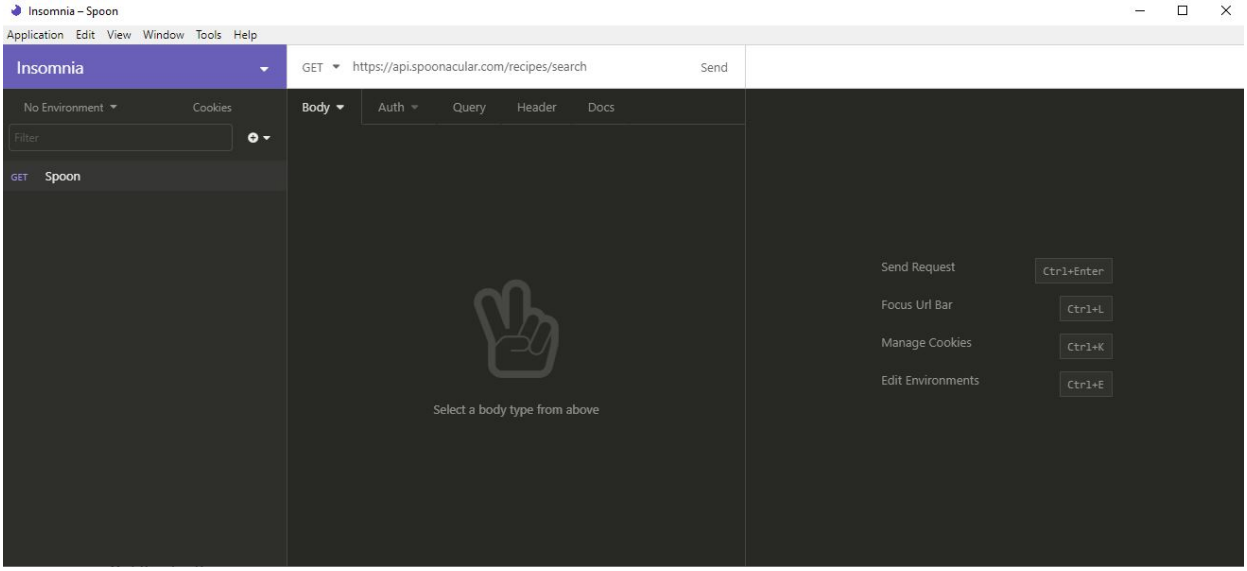
To illustrate how to send requests outside of the web browser, we will again use the spoonacular API. However, this time around, we will use Insomnia. This software allows us to easily construct requests, as well as returns responses and auto-formats them. This is not a tool to be used in production environments. However, it is very useful for when we are writing code in a language (like R). When we write code to auto generate the requests, we need to have an idea of the request structure. Hence, Insomnia helps us towards this objective of understanding. So let's get started. First, load up Insomnia, and you should get this screen:



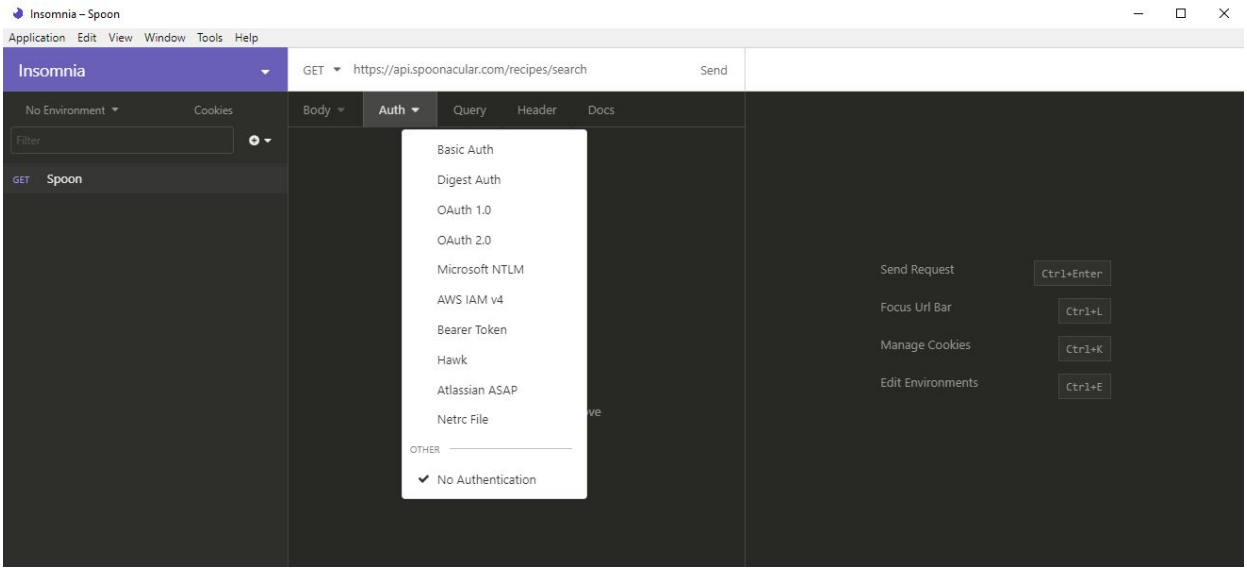
We are going to create a new request. Click on "New Request", which should bring up the screen below. Give the request a name (any name will suffice), and ensure it is marked as "GET" on the right hand side of the drop down. Why GET? Because the spoonacular documentation tells us that all the requests are GET requests. Click "Create":



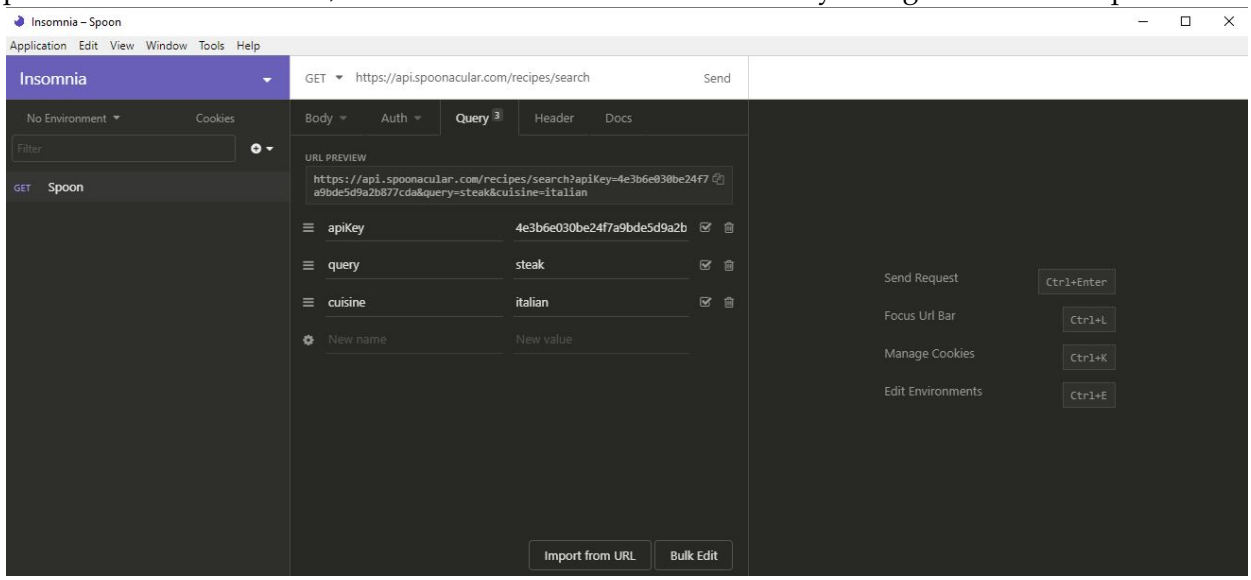
Now we are ready to construct our request. First, we will specify the url to the endpoint we want to call. In the url, next to "GET", put in the url for the endpoint of the API. We want to search for recipes again, so we will use the endpoint `https://api.spoonacular.com/recipes/search`



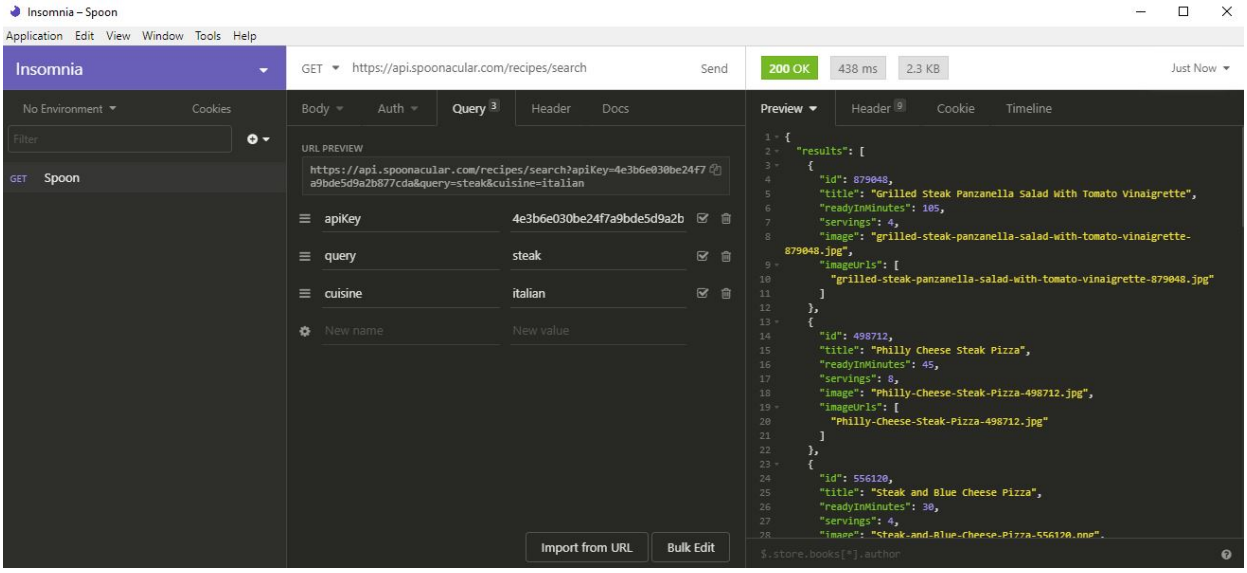
First, according to the documentation, we do not need to construct a body in our request (this will change when we get to Yelp!). All parameters are passed in the url. Hence, leave the "Body" portion alone. Move to "Auth", and ensure that "No Authentication" is clicked. This may sound odd, given that spoonacular uses authentication. However, when authentication is used in the url string, we always mark "No Authentication" in this screen:



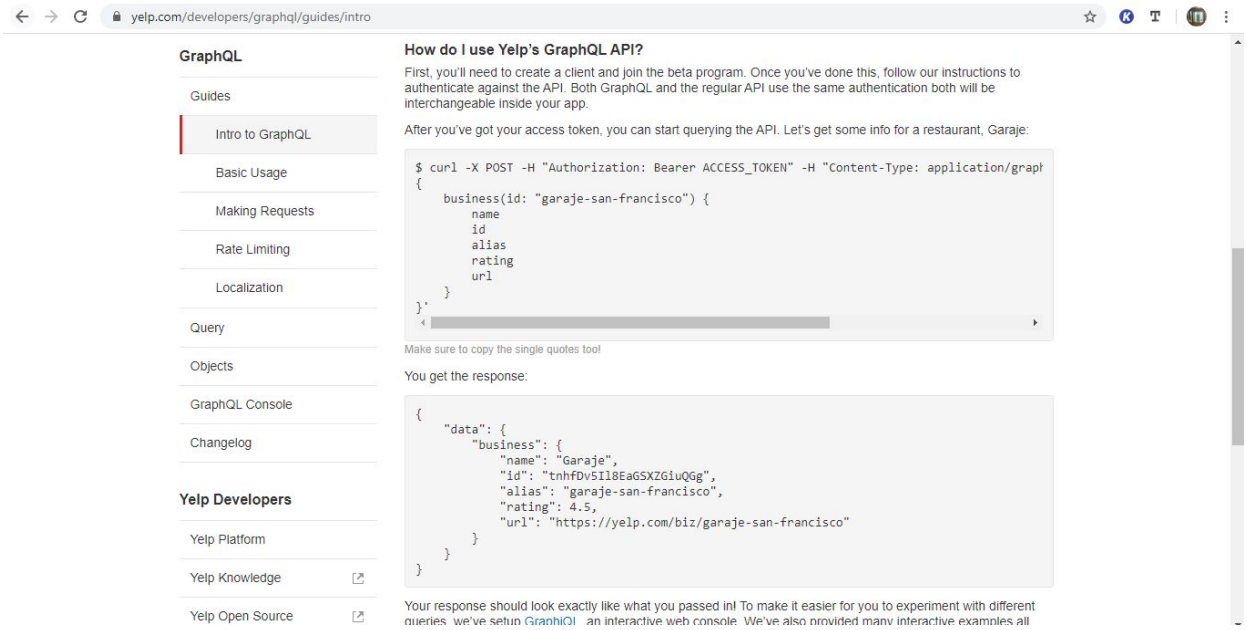
Now we need to construct the parameters. Recall that for spoonacular, we authenticate ourselves by providing an apiKey parameter. Add it to the parameter list as shown below. Recall that mine will be different than yours, so ensure to go back to your account to get the right key. In addition, we will search this time for steak recipes that fall under the "italian" cuisine category. According to the documentation, we can search for "steak" by providing this to the "query" parameter. Furthermore, we can search for italian cuisine by using the "cuisine" parameter:



At this point, we have nothing to put in our request header, and so we are done! Click "Send" and let's get a response:

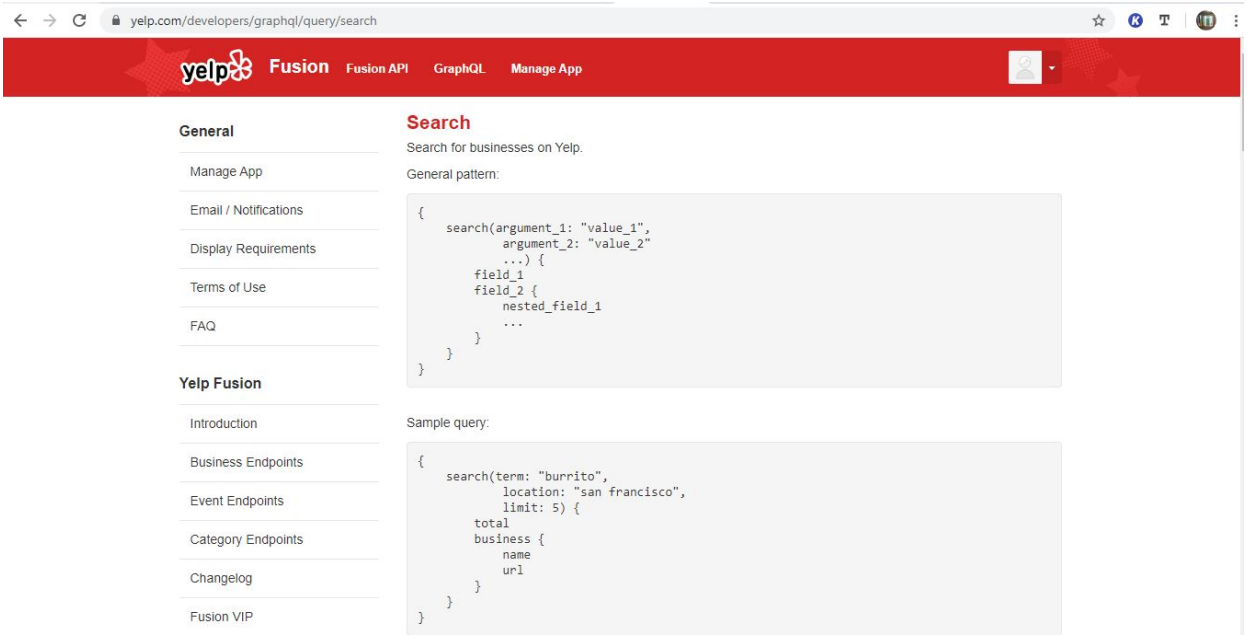


Congrats! You have made your first request to an API without using a browser! Now time to get things more, interesting! Let us work on the Yelp! API. With a different type of Authentication. Unlike previous uses of APIs up to this point, the Yelp! API uses a different type of Authentication method that does not rely on a parameter passed through the url string. Instead, you need what is called a BEARER_TOKEN. We can find this information by going to the documentation for the Yelp! API:

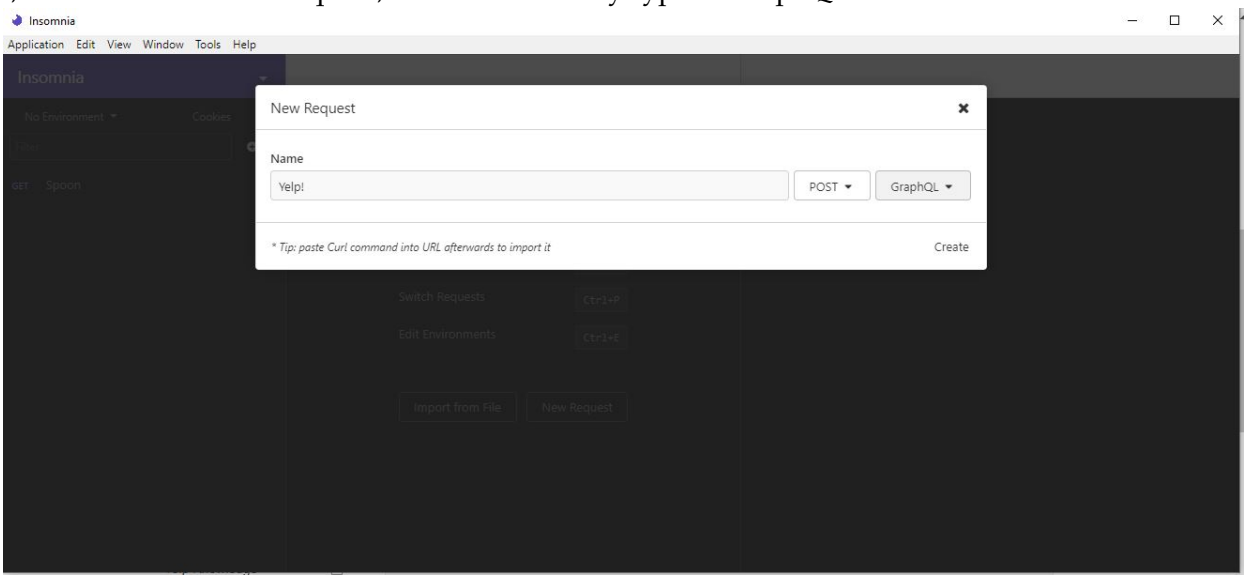


One thing of note. Yelp! has two different types of APIs. First, they have a traditional API (the one we have been using up to this point), with the idea of endpoints (This is the Yelp! Fusion API). The second type of API they have is called a GraphQL API. This type of API works slightly different. Instead of GET requests with queries, the user sends a POST request with JSON code about the specific type of information they want the server to return. I will leave it to you to read through the documentation on this. However, the basic idea is to specify in the JSON code the objects and the properties of the objects you want returned. The documentation has examples

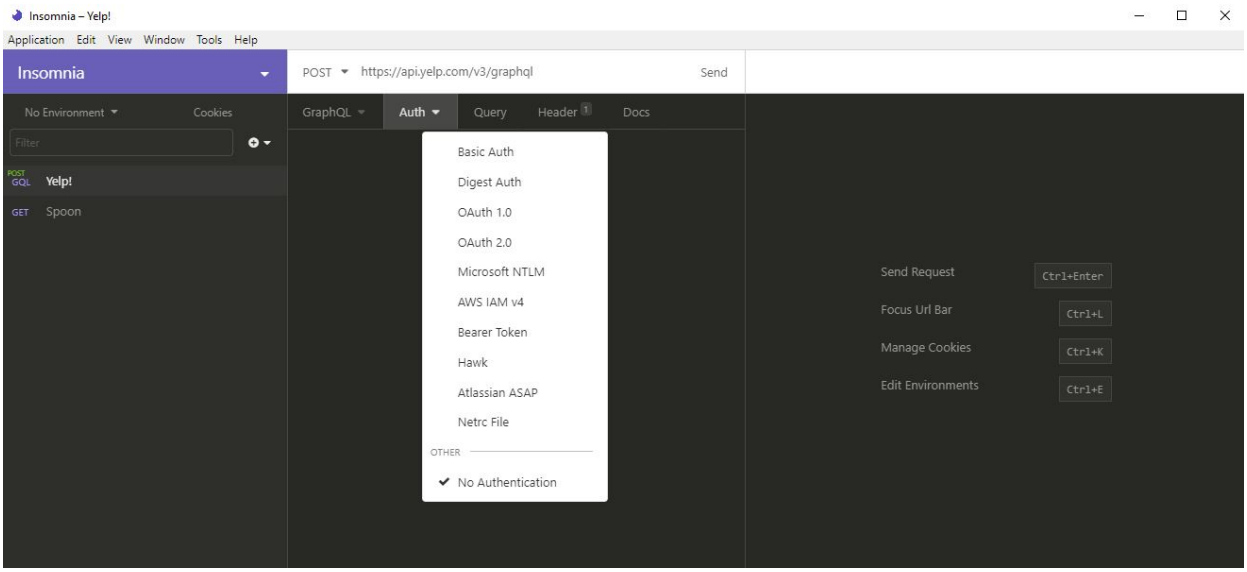
and information on each and every object available. For example, in order to search on the GraphQL, Yelp! has the following information:



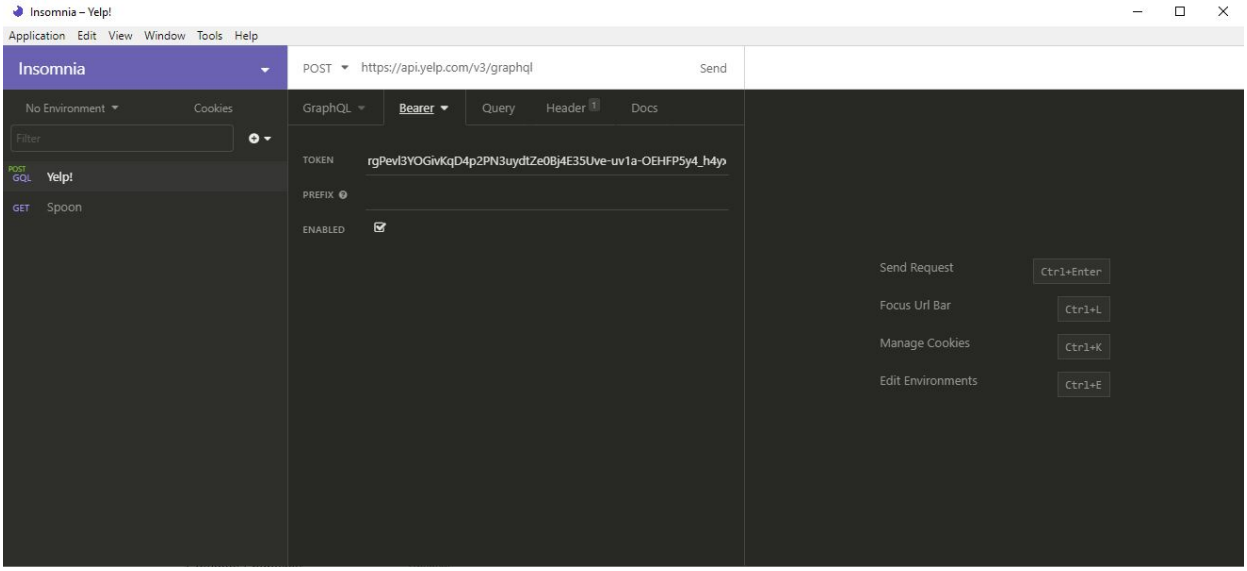
Now, lets use the GraphQL to make requests (by the way, other APIs also use a GraphQL. Whichever you decide to use is your choice). Open Insomnia and create a new request. This time, ensure it is a POST request, and that the body type is GraphQL:



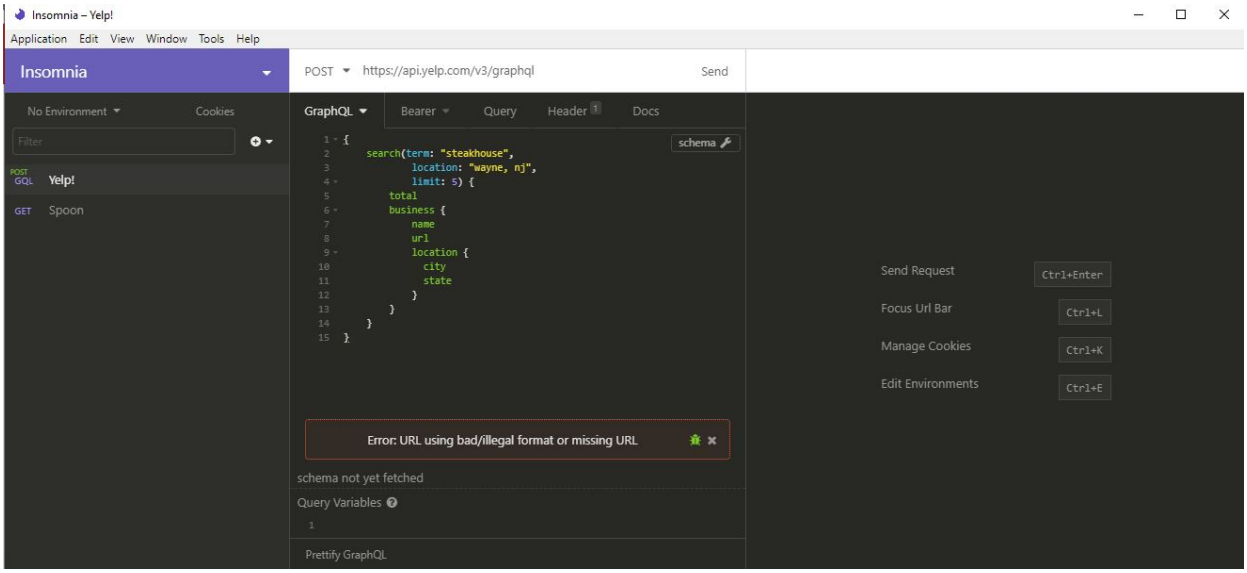
Add in the endpoint to the GraphQL, which for Yelp! is `https://api.yelp.com/v3/graphql`. Also, click the drop down box for "Auth", and click on "Bearer Token":



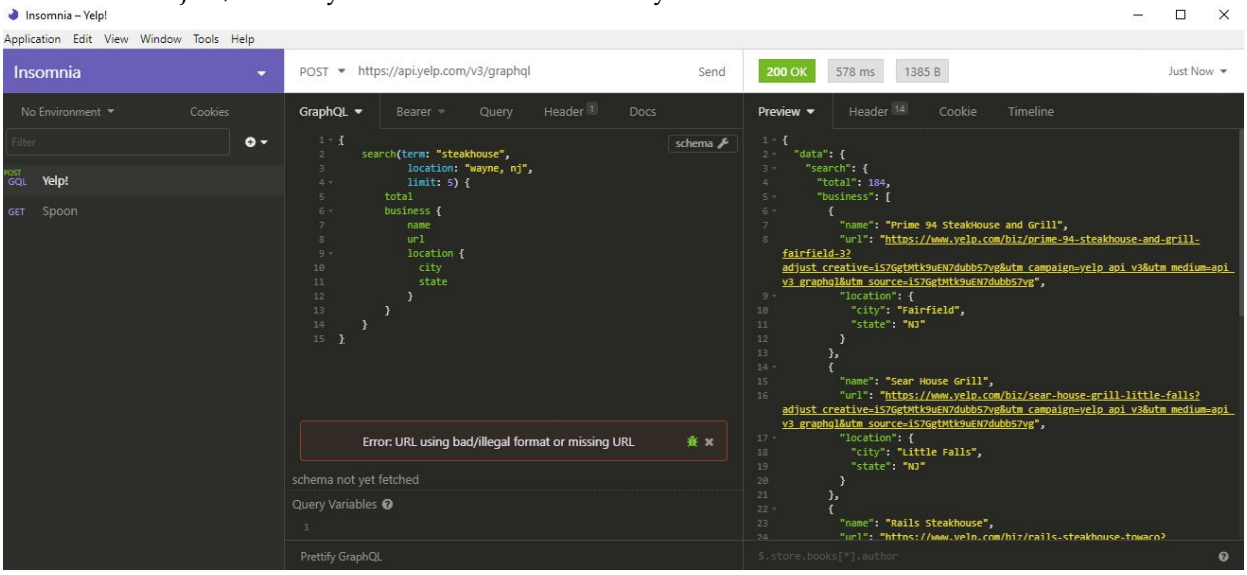
Now, input the token id from your Yelp! "Manage App" screen (see earlier in the tutorial, I TOLD YOU TO SAVE THIS!):



Finally, onto the query. Click on the "GraphQL" tab, and insert the JSON shown below. Click "Send":



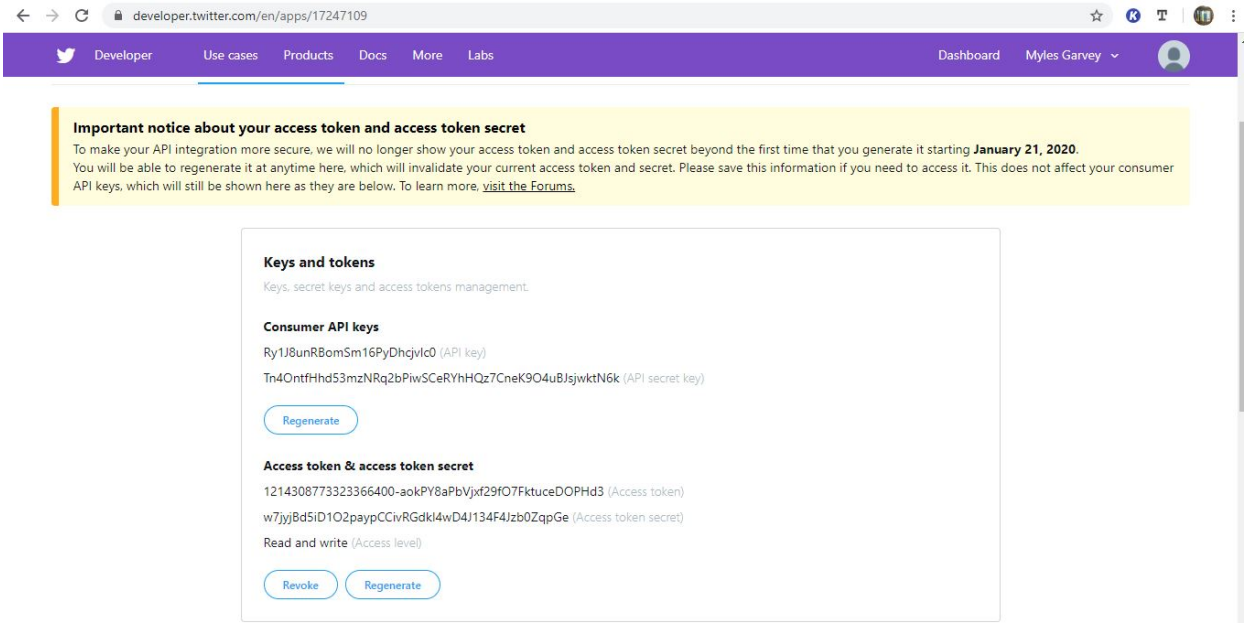
Finally, we have out output show. Notice the output that is showing. Unlike endpoints in traditional APIs, the GraphQL returned back to us only the information we requested. In the query posted to the server specified by the JSON code, we are sending a "search" query, specifying parameters such as the term to search for, the location to search within, and the number of responses to send back. These are parameters specified in the search query documentation. Next, notice that we want the GraphQL to return the total number of results, and a list of businesses. For each business, we want it to only return the business name, the url, and the location object. In the location object, we only want it to return the city and the state:



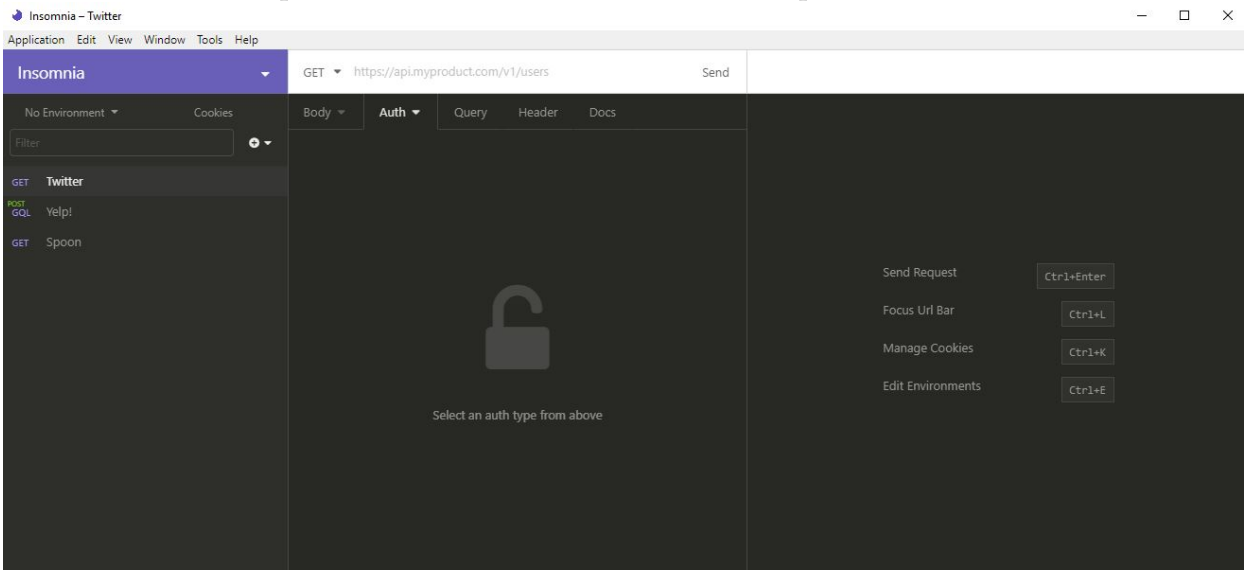
Looking to the result of the first business in the list, we notice that GraphQL sent back information only on what we asked for. Obviously, we could have asked for more (as indicated by the documentation guides). This is why GraphQL is gaining popularity as an API tool. It allows us to get only the information we need, rather than ALL the information about objects available via an API. This is a much more efficient way of querying information to an API.

Last, we will move onto using Twitter. Recall that Twitter uses OAuth. This means we need to provide 4 "tokens". Recall that the Twitter API did this for us (go back to the set up section of

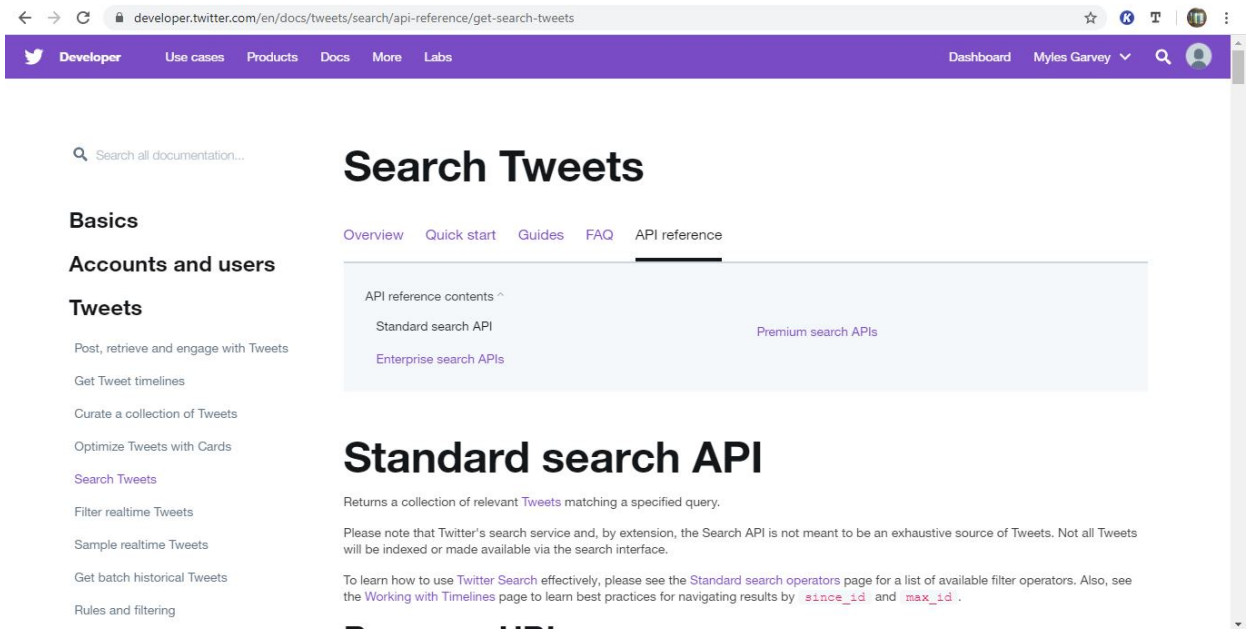
this tutorial) when we signed up and generated our keys and tokens. Go back to this page if you have not already to get your specific tokens and keys:



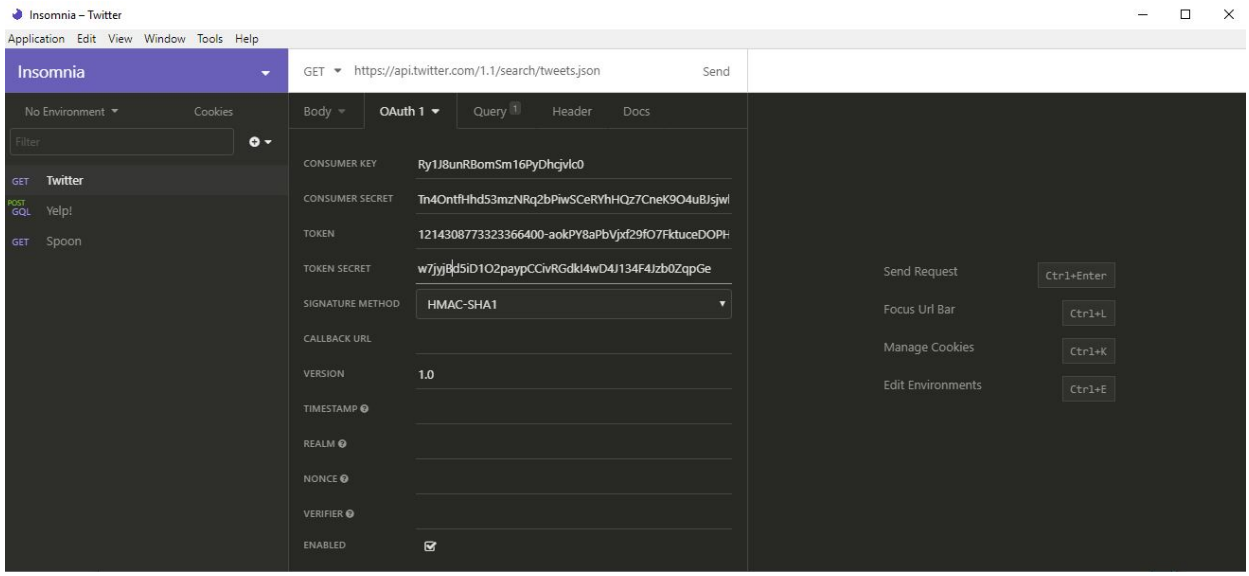
Now, create a new request in Insomnia. Ensure it is a GET request:



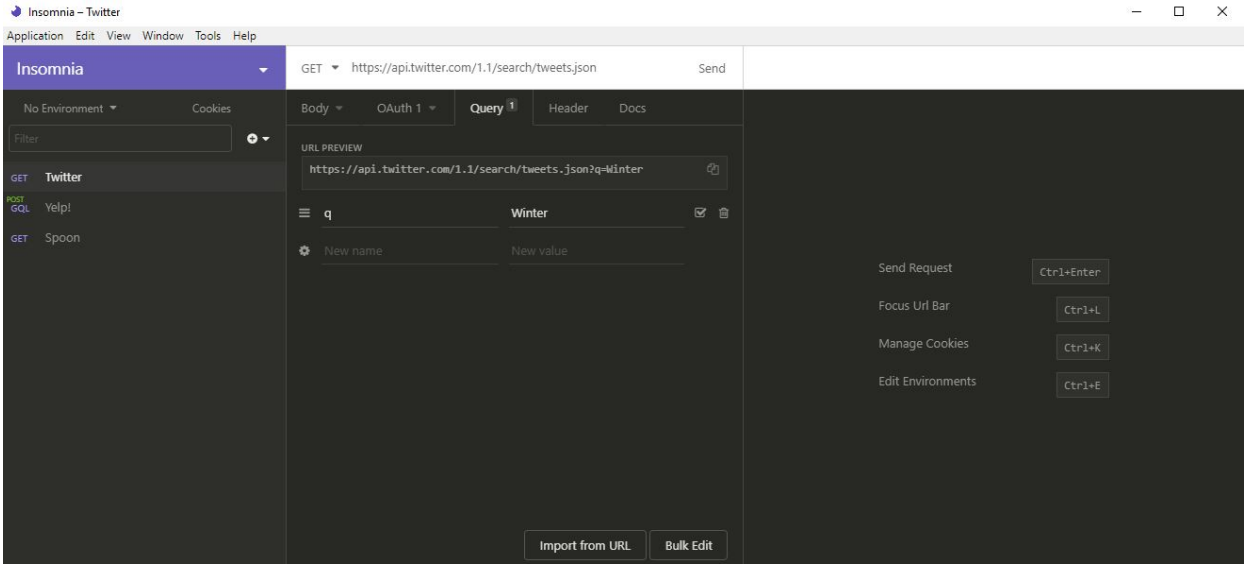
We will be using the Search Tweets endpoint in the Twitter API. You can find the documentation for it here:



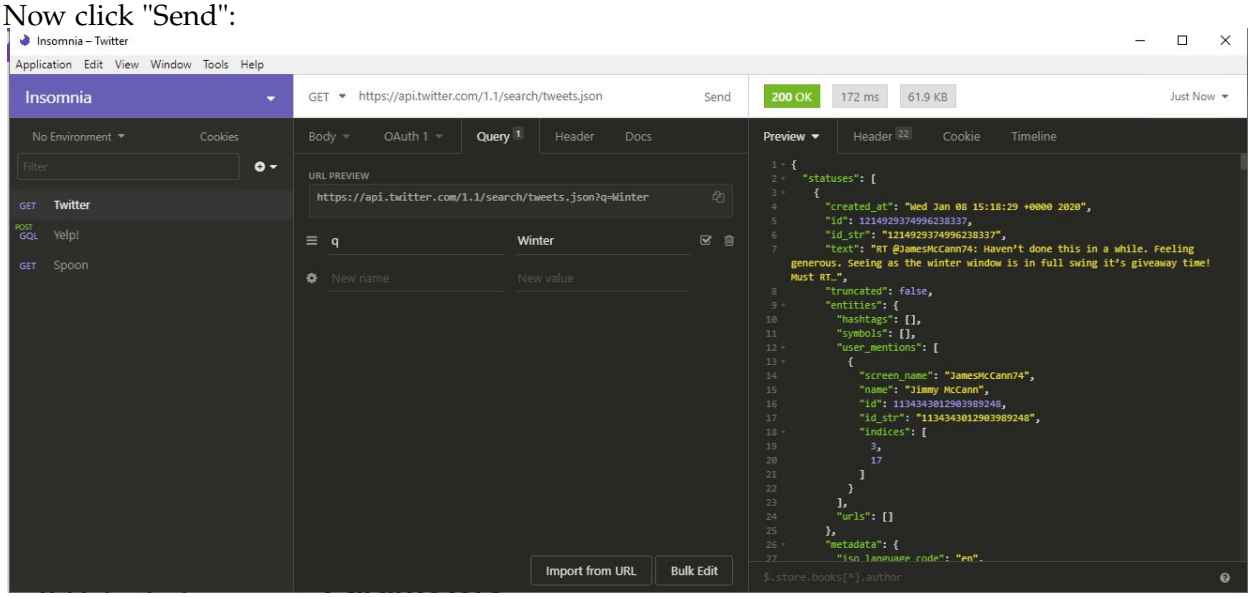
In Insomnia, click on OAuth 1 for the authentication method. Using the keys and tokens from earlier, put all 4 into the 4 respective boxes in this screen:



Now, if you read the documentation for this endpoint, you will know its inputs. One input is "q", which is the search term you want to search across Twitter. Here, we will search for Tweets that have the term "Winter" in them:



Now click "Send":



WARNING: If you get an authorization or authentication error, it means you most likely have (1) the incorrect keys and tokens, or (2) accidentally put a space. Make sure there are NO spaces at the end of the input! I made this same mistake, so don't let this drive you nuts for hours, and just double check there are no spaces in these fields.

Now, the output of the search endpoint is of course a collection of Tweets. Each Tweet has over 80 pieces of information associated with it. This is a lot of information, some useful, some excessive. Up to this point, we have been having JSON returned to us from all of these APIs. While this data format is very useful to represent multi-dimensional information, we still up to this point have not processed any of the information in R yet. As I mentioned, Insomnia is useful for testing queries, but is not so much for processing the output. So, let us see how we can access APIs, and process the output, in R.

There are many ways to do this in R, as we shall see in a moment. One of the most popular, easiest, and efficient way of accessing APIs is by constructing requests using the `httr` package in R. We will illustrate how to use this with the Twitter API. Suppose we want to find all Tweets with

the keyword "Winter". As always, when constructing a response, we need to find the following info and some how construct the response with the info. Here is the info for this request, which the API Documentation was used to determine these properties:

- Authorization: OAuth 1.0. Here we will need to provide 4 tokens, namely a customer key and secret, as well as an access key and secret, just as we did before in Insomnia.
- Endpoint URL: We are using the Search endpoint, which the API Documentation tells us has the URL
  `https://api.twitter.com/1.1/search/tweets.json`
- Parameters: `q`. We could use others, but all we want is to search for the term "Winter". Hence, we need to pass a parameter to the url string with `q=Winter`.

To do this in R, we need to use `httr` package's functions to first construct an authentication portion of the request, which we will construct as what is called a *token*. The token essentially will be a list in R that has information regarding the authentication which the request will use to send it to the server. After we create a token, we need to create our url, as usual, by first giving the url to the api, then the url to the endpoint, and last followed by a list of `parameter=value` entries. The url can be constructed in many ways in R. We will just simply assign the url variable to a string with the parameters manually entered into the url. We then use the function that bears the name of the type of request we seek to use (in this case, GET), passing in the url (which carries all the request information) and the token (which carries all the authentication information):

```
> library(httr)
>
> #Create OAuth Token (Customer Key and Secret)
> app <- oauth_app("twitter",
+                  key="Ry1J8unRBomSm16PyDhcjvlc0",
+                  secret = "Tn4OntfHhd53mzNRq2bPiwSCeRYhHQz7CneK9O4uBJsjwktN6k"
   )
>
> #Create OAuth Credentials (Access Key and Secret)
> credentials <- list(oauth_token = "1214308773323366400-
   aokPY8aPbVjxf29fO7FktuceDOPHd3",
+                      oauth_token_secret = "
   w7jyjBd5iD1O2paypCCivRGdkI4wD4J134F4Jzb0ZqpGe")
>
> #Ensure that we put this in the header of the request
> params <- list(as_header = TRUE)
>
> #Create the token with the twitter endpoints
> token <- Token1.0$new(app = app,
+                       endpoint = oauth_endpoints("twitter"),
+                       params = params,
+                       credentials = credentials,
+                       cache = FALSE)
> #Endpoint URL:
> eURL<-"https://api.twitter.com/1.1/search/tweets.json?q=Winter"
>
> #Send the GET Rquest
> req <- GET(eURL,token)
```

```
>
> #Convert any http request errors into R errors
> stop_for_status(req)
>
> #Grab the Response Data.  R will return it as a list rather than in JSON
> response<-content(req)
>
> #See the structure of the response:
> names(response)
[1] "statuses"        "search_metadata"
>
> #We only care about the Twitter statuses:
> statuses<-response$statuses
>
> #Take A Peek Inside Some of The Properties of the First Tweet:
> names(statuses[[1]])
 [1] "created_at"                "id"
 [3] "id_str"                    "text"
 [5] "truncated"                 "entities"
 [7] "extended_entities"         "metadata"
 [9] "source"                    "in_reply_to_status_id"
[11] "in_reply_to_status_id_str" "in_reply_to_user_id"
[13] "in_reply_to_user_id_str"   "in_reply_to_screen_name"
[15] "user"                      "geo"
[17] "coordinates"               "place"
[19] "contributors"              "is_quote_status"
[21] "retweet_count"             "favorite_count"
[23] "favorited"                 "retweeted"
[25] "possibly_sensitive"        "lang"
> statuses[[1]][c("created_at","text","retweeted")]
$created_at
[1] "Thu Jan 09 12:06:27 +0000 2020"

$text
[1] "So yeah  i t s  winter but  t h e r e s  no reason for being this cold \U0001f976
    (-13 degrees fahrenheit) https://t.co/3CkLjxo6TI"

$retweeted
[1] FALSE
```

Listing 7: Create Authentication Tokena nd GET Request for Twitter API.

After gathering the data, we can also collect, say, all retweet data to analyze. Notice however that by default, the API only returns 15 tweets per "page". You can increase this to 100 by using the `count` parameter:

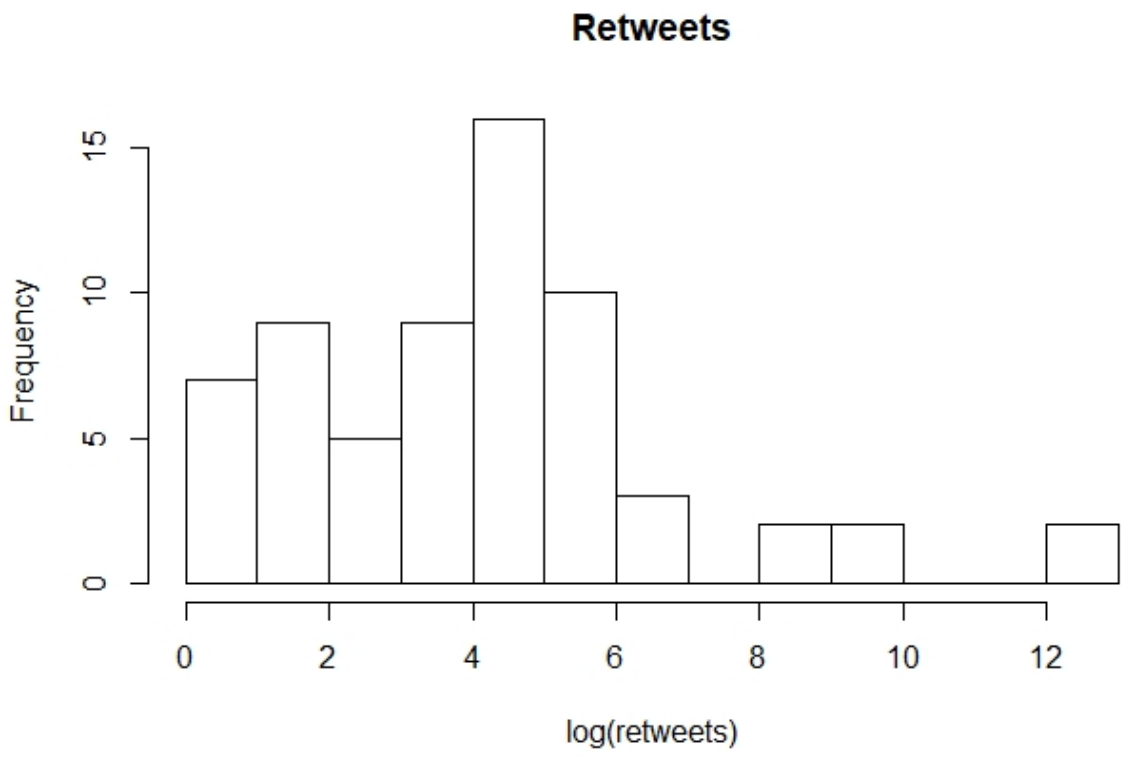
```
> #Endpoint URL:
> eURL<-"https://api.twitter.com/1.1/search/tweets.json?q=Winter&count=100"
>
> #Send the GET Rquest
> req <- GET(eURL,token)
>
> #Convert any http request errors into R errors
> stop_for_status(req)
>
> #Grab the Response Data.  R will return it as a list rather than in JSON
```

```
> response<-content(req)
>
> #See the structure of the response:
> names(response)
[1] "statuses"        "search_metadata"
>
> #We only care about the Twitter statuses:
> statuses<-response$statuses
>
> #Use lapply to iterate through each tweet in the list, and have it pull only
> #the retweet_count property from the tweet.  Then, unlist it to conver it to
> #a vector of retweets.
> retweets<-unlist(lapply(statuses,function(x){x$retweet_count}))
>
> summary(retweets)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
     0.0      0.0      6.5   9040.2    110.8 433008.0

> hist(log(retweets),main="Retweets",breaks=10,outline=FALSE)
```

Listing 8: Changing the number of tweets to return to 100

If we take the log of this data (which helps us smooth out our outliers without fundamentally changing the structure of our data), we have:



Now let us try to do this with the Yelp! API. Like before with Twitter, we need to specify the authentication in the header of the request (just as we do with OAuth 1.0). The Authentication Method here is BEARER TOKEN, just as we had seen before. However, instead of using the GraphQL, we will use the traditional Fusion API, which has the same functionality, but will

return all data about businesses rather than just a handful:

```
> #Create and Authentication Header
> auth_header <- add_headers('Authorization'= paste('Bearer',apiKey))
>
> #Construct the URL.  Recommended to use Insomnia first to ensure spaces are
    proper
> url<-"https://api.yelp.com/v3/businesses/search?term=Pizza&location=Wayne,%20NJ"
>
> #Send it off!
> rep<-GET(url,auth_header)
>
> #Grab Response Data:
> body_rep<-content(rep)
>
> #Inspect Structure:
> names(body_rep)
[1] "businesses" "total"      "region"
>
> #Inspect Business Structure:
> names(body_rep[["businesses"]][[1]])
 [1] "id"           "alias"        "name"         "image_url"    "is_closed"
 [6] "url"          "review_count" "categories"   "rating"       "coordinates"
[11] "transactions" "price"        "location"     "phone"        "display_
    phone"
[16] "distance"
>
> #Get all business names that returned
> b_names<-unlist(lapply(body_rep[["businesses"]],function(x){x$name}))
> b_names
 [1] "Pizza One"
 [2] "Pie Guys Pizza"
 [3] "Neil's Pizzeria"
 [4] "Coney Island Wood Fired Pizza"
 [5] "Vinni's Pizzarama"
 [6] "Buonocores brick oven pizza, panini, and mozzer"
 [7] "Positano Restaurant & Pizzeria"
 [8] "Amore's Pizza & Pasta"
 [9] "Buongusto Pizza Restaurant & Catering"
[10] "Dominick's Pizzeria"
[11] "Se7te Wood Fire Pizza"
[12] "Uncle Louie's Pizza"
[13] "Gencarelli's"
[14] "Taste Of Tuscany"
[15] "Aquila Pizza Al Forno"
[16] "Time Out Cafe"
[17] "Brother Bruno's Pizza, Deli & Bagels"
[18] "Bradlees Pizzeria"
[19] "Anthony's Coal Fired Pizza"
[20] "Gencarelli's"
```

Listing 9: Using the Yelp! API in R.

Let us look at how to use the GraphQL as well. Recall with GraphQL, instead of sending a GET request, we send a POST request, where the body of the message is a JSON query for specific information. Instead of having different endpoints, the API will look at the text in your JSON

query, determine the appropriate endpoint, and only get and return specific request information, rather than all the information. Doing this is a more efficient way of downloading information from an API, since it will only return what we want. Let us repeat our query to Yelp! as we did in Insomnia, with the exception being of searching for universities:

```
>#Create and Authentication Header.  Note that we also change the Content-Type
    parameter in the header:
> auth_header <- add_headers('Authorization'= paste('Bearer',apiKey),
+                            'Content-Type' = "application/graphql"
+                            )
>
> #Construct the URL.  This is just the endpoint for the graphql on the server
> url<- "https://api.yelp.com/v3/graphql"
>
> #Construct the GraphQL Query in JSON:
> body<-"{
+     search(term: \"universities\",
+             location: \"wayne, nj\",
+             limit: 5) {
+         total
+         business {
+             name
+             url
+             location {
+               city
+               state
+             }
+         }
+     }
+ }"
>
>
> #Send it off!
> rep<-POST(url,auth_header,body=body)
>
> #Grab Response Data:
> body_rep<-content(rep)
>
> #Inspect Structure:
> names(body_rep)
[1] "data"
>
> #Inspect data Structure:
> names(body_rep[["data"]])
[1] "search"
>
> #Inspect search structure:
> names(body_rep[["data"]][["search"]])
[1] "total"    "business"
>
> #Get list of businesses:
> businesses<-body_rep[["data"]][["search"]][["business"]]
>
> #Inspect properties of single business:
> names(businesses[[1]])
```

```
[1] "name"     "url"      "location"
>
> #Get all business names and cities that returned
> b_names<-unlist(lapply(businesses,function(x){c(x$name)}))
> b_urls<-unlist(lapply(businesses,function(x){c(x$location$city)}))
>
> #Now construct a data frame with our data that we pulled from the API:
> businesses<-data.frame(business_name = b_names,city = b_urls)
>
> businesses
                business_name      city
1 William Paterson University     Wayne
2  Montclair State University Montclair
3            Caldwell College  Caldwell
4       C2 Education of Wayne     Wayne
5         Caldwell University  Caldwell
```

Listing 10: Doing a GraphQL Query in Yelp!.

To generalize this process, again, we must think about how authentication happens on the API end. You already now know how to do this in R with OAuth 1.0, BEARER TOKEN, and no authentication. If an API uses a different type of authentication, you will of course need to determine how to add this into your request in R. Typically, it is included in some way in the request header, which we illustrated above, can be changed/added to by using the `add_header` function in R. Upon reading the documentation, you must also determine if an API necessitates GET or POST requests (and sometimes, they use other types as well).

To end our workshop, we will review through a simplification of the Twitter API. Luckily, there has been a simplification of the Twitter API in R. Two packages exist which simplify the process of requesting information from Twitter. We will discuss only one of them, namely `rtweet`. The `rtweet` package in R acts as a cover to the entire API, which basically converts inputs into R functions into the requests, sends them out, and receives the Tweet information, and last packages up the information into a Tibble (which can easily be converted to a data frame). To use the package, we first need to create an authentication token (which also is simplified). After the token is created, we can start to use some of the functions in RTweet. Let us illustrating a Tweet search of the word "Winter" using the `rtweet` package:

```
> library(rtweet)
> #Create the Token
> token<- create_token(
+   app="BANDMDW",
+   consumer_key="Ry1J8unRBomSm16PyDhcjvlc0",
+   consumer_secret = "Tn4OntfHhd53mzNRq2bPiwSCeRYhHQz7CneK9O4uBJsjwktN6k",
+   access_token = "1214308773323366400-aokPY8aPbVjxf29fO7FktuceDOPHd3",
+   access_secret= "w7jyjBd5iD1O2paypCCivRGdkI4wD4J134F4Jzb0ZqpGe")
>
> #Search for tweets with the keyword "Winter".
> tweets<-search_tweets("Winter",token=token)
>
> #Convert to a data frame
> tweets<-data.frame(tweets)
>
> #Inspect the names in the data frame
```

```
> names(tweets)
 [1] "user_id"                 "status_id"
 [3] "created_at"              "screen_name"
 [5] "text"                    "source"
 [7] "display_text_width"      "reply_to_status_id"
 [9] "reply_to_user_id"        "reply_to_screen_name"
[11] "is_quote"                "is_retweet"
[13] "favorite_count"          "retweet_count"
[15] "quote_count"             "reply_count"
[17] "hashtags"                "symbols"
[19] "urls_url"                "urls_t.co"
[21] "urls_expanded_url"       "media_url"
[23] "media_t.co"              "media_expanded_url"
[25] "media_type"              "ext_media_url"
[27] "ext_media_t.co"          "ext_media_expanded_url"
[29] "ext_media_type"          "mentions_user_id"
[31] "mentions_screen_name"    "lang"
[33] "quoted_status_id"        "quoted_text"
[35] "quoted_created_at"       "quoted_source"
[37] "quoted_favorite_count"   "quoted_retweet_count"
[39] "quoted_user_id"          "quoted_screen_name"
[41] "quoted_name"             "quoted_followers_count"
[43] "quoted_friends_count"    "quoted_statuses_count"
[45] "quoted_location"         "quoted_description"
[47] "quoted_verified"         "retweet_status_id"
[49] "retweet_text"            "retweet_created_at"
[51] "retweet_source"          "retweet_favorite_count"
[53] "retweet_retweet_count"   "retweet_user_id"
[55] "retweet_screen_name"     "retweet_name"
[57] "retweet_followers_count" "retweet_friends_count"
[59] "retweet_statuses_count"  "retweet_location"
[61] "retweet_description"     "retweet_verified"
[63] "place_url"               "place_name"
[65] "place_full_name"         "place_type"
[67] "country"                 "country_code"
[69] "geo_coords"              "coords_coords"
[71] "bbox_coords"             "status_url"
[73] "name"                    "location"
[75] "description"             "url"
[77] "protected"               "followers_count"
[79] "friends_count"           "listed_count"
[81] "statuses_count"          "favourites_count"
[83] "account_created_at"      "verified"
[85] "profile_url"             "profile_expanded_url"
[87] "account_lang"            "profile_banner_url"
[89] "profile_background_url"  "profile_image_url"
>
> #Only grab the screen name, tweet creation timestamp, and retweet count for each
    tweet returned
> tweets[,c("screen_name","created_at","retweet_count")]
        screen_name          created_at retweet_count
1     eleanorquiroz 2020-01-09 13:40:32             6
2         Annexcali 2020-01-09 13:40:31             0
3          clapa238 2020-01-09 13:40:31             1
4       ShanghaiEye 2020-01-09 13:40:31             0
5      winter_fuego 2020-01-09 13:40:30          3164
```

```
6   alloneworddotie 2020-01-09 13:40:30             0
7           ymzrspkc 2020-01-09 13:40:30             0
8        MarcJaylynn 2020-01-09 13:40:30             2
9          S3roTonyn 2020-01-09 13:40:29             0
10       ChSuptRoads 2020-01-09 13:40:28             7
11    RHeightsFinest 2020-01-09 13:40:27             0
12         jbonnyman 2020-01-09 13:40:27             5
13        cutie_icee 2020-01-09 13:40:27            78
14     citybhlight67 2020-01-09 13:40:27            14
15    winter_flower3 2020-01-09 13:40:26             0
16     friendoftoads 2020-01-09 13:40:26             0
17         LOLO_0526 2020-01-09 13:40:26            38
18    SummerBreezeUS 2020-01-09 13:40:25             1
19         pan411170 2020-01-09 13:40:25            11
20      zwaantje1983 2020-01-09 13:40:24             0
21     _winter_baby_ 2020-01-09 13:40:24            24
22      to_themoon_0 2020-01-09 13:40:24          3905
23        BTSNJY1230 2020-01-09 13:40:23            13
24         PLaosupho 2020-01-09 13:40:23           159
25         Cxo_diana 2020-01-09 13:40:22          9060
26  robertmacIntos3 2020-01-09 13:40:22            20
27       JKHOPE_twt 2020-01-09 13:40:21            10
28     GlenviewILPD 2020-01-09 13:40:21             0
29  cH06SNWXbhU30pu 2020-01-09 13:40:20           217
30         uwhie_RCL 2020-01-09 13:40:20        433123
31    BABY_LEO_BILL 2020-01-09 13:40:20           180
32    lemonSVTcarat 2020-01-09 13:40:19            72
33      HOBlHOBl_S2 2020-01-09 13:40:19             0
34     DiscoverNEPA 2020-01-09 13:40:19             0
35        donuts6666 2020-01-09 13:40:18            67
36        Bom_Winter 2020-01-09 13:40:18            33
37     Aarti_Panday 2020-01-09 13:40:18            69
38         Vibezzz__ 2020-01-09 13:40:17          3068
39    pcsendashonga 2020-01-09 13:40:17             5
40     Kat_Missouri 2020-01-09 13:40:17             4
41    Faust89823700 2020-01-09 13:40:15             3
42   IrisOfTheBronx 2020-01-09 13:40:15             2
43      markwinder8 2020-01-09 13:40:15             1
44  GrandsonofRickk 2020-01-09 13:40:14           201
45      Ariluminosa 2020-01-09 13:40:14             0
46   Empress_Marsha 2020-01-09 13:40:13             0
47     mistrasparta 2020-01-09 13:40:13             0
48   AutosourcePtbo 2020-01-09 13:40:13             0
49        eeaaakkkk 2020-01-09 13:40:13             0
50          ggukmei 2020-01-09 13:40:12          5532
51  hosoku_is_mylif 2020-01-09 13:40:11            31
52   maxtheultracat 2020-01-09 13:40:10             0
53   puadekplaylist 2020-01-09 13:40:09           268
54   OyekanmiCollar 2020-01-09 13:40:08             3
55  XRP_589_Theunis 2020-01-09 13:40:08             1
56   Leonor93726311 2020-01-09 13:40:08          1522
57          1stEdCU 2020-01-09 13:40:08             0
58        morrelife 2020-01-09 13:40:08           581
59    winter_pokeca 2020-01-09 13:40:06             0
60         hrdrock2 2020-01-09 13:40:05            67
61  8AvLFe5JwBhiCOm 2020-01-09 13:40:05          3905
```

```
128 62           Imskey 2020-01-09 13:40:05             0
129 63   AndrewRussel15 2020-01-09 13:40:05             1
130 64      AeroArrowAS 2020-01-09 13:40:04          3456
131 65       MacNomadic 2020-01-09 13:40:04             2
132 66  LouisChicharit4 2020-01-09 13:40:04           581
133 67  HockomockSports 2020-01-09 13:40:04             0
134 68     ConnGardener 2020-01-09 13:40:04             0
135 69      winter_floo 2020-01-09 13:40:04           333
136 70          StckPro 2020-01-09 13:40:03             0
137 71    MrCoachWilson 2020-01-09 13:40:03             3
138 72    onemoredayplz 2020-01-09 13:40:03             3
139 73          judah47 2020-01-09 13:40:03             0
140 74     Winter_Socks 2020-01-09 13:40:03           492
141 75    Winter_Lalala 2020-01-09 13:40:03            85
142 76  AkiraTakahikaru 2020-01-09 13:40:03           603
143 77  Frugal__Finance 2020-01-09 13:40:02             0
144 78     Joker_Winter 2020-01-09 13:40:02             0
145 79       mio_winter 2020-01-09 13:40:02             0
146 80           joyhog 2020-01-09 13:40:01             0
147 81    a_lg_dubldubl 2020-01-09 13:40:01             0
148 82   MorrisJonathan 2020-01-09 13:40:01             0
149 83        8_Bit_Era 2020-01-09 13:40:00             0
150 84     yegwxnerdery 2020-01-09 13:40:00             0
151 85     yegwxnerdery 2020-01-09 13:39:58             0
152 86     Variasion_Co 2020-01-09 13:39:59            10
153 87      La_Pintora1 2020-01-09 13:39:59             4
154 88             pKjd 2020-01-09 13:39:59             0
155 89   rhasastika0515 2020-01-09 13:39:59           444
156 90       imjustdmal 2020-01-09 13:39:59             1
157 91      TheAntRider 2020-01-09 13:39:59             0
158 92  VanTAEholic1995 2020-01-09 13:39:59           651
159 93        gerdaikai 2020-01-09 13:39:58             0
160 94         Poopk613 2020-01-09 13:39:58           201
161 95        perjoste1 2020-01-09 13:39:58            22
162 96     hassouuuniii 2020-01-09 13:39:58             3
163 97       mkbaumbach 2020-01-09 13:39:57             4
164 98    almondchoc_gb 2020-01-09 13:39:57          1844
165 99         kaosonia 2020-01-09 13:39:56            11
166 100    Ana___Winter 2020-01-09 13:39:56             0
```

Listing 11: Searching for Tweets with Winter using RTweet.

As we can see, the `rtweet` package greatly simplifies our work when trying to extract data. However, not every API has an easy to use package in R, which is why we needed to generalize our discussion to the mechanics of connecting and requesting information. With this stated, you now should be able to download anything from any API with public access using R! It is of no doubt that this is an important skill to craft, since accessing data in APIs greatly reduces your data mining and analysis.

Exercises

5. Search in the Yelp! API for 15 businesses in the Paterson, NJ location that are supermarkets. Use GraphQL in R for Yelp! to do this.

6. Use the `rtweet` package that will stream live tweets for 5 seconds.

7. Use the `rtweet` package that will download all available tweets for the Twitter screenname "wpunj_edu".