

# 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 programmatically.

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.

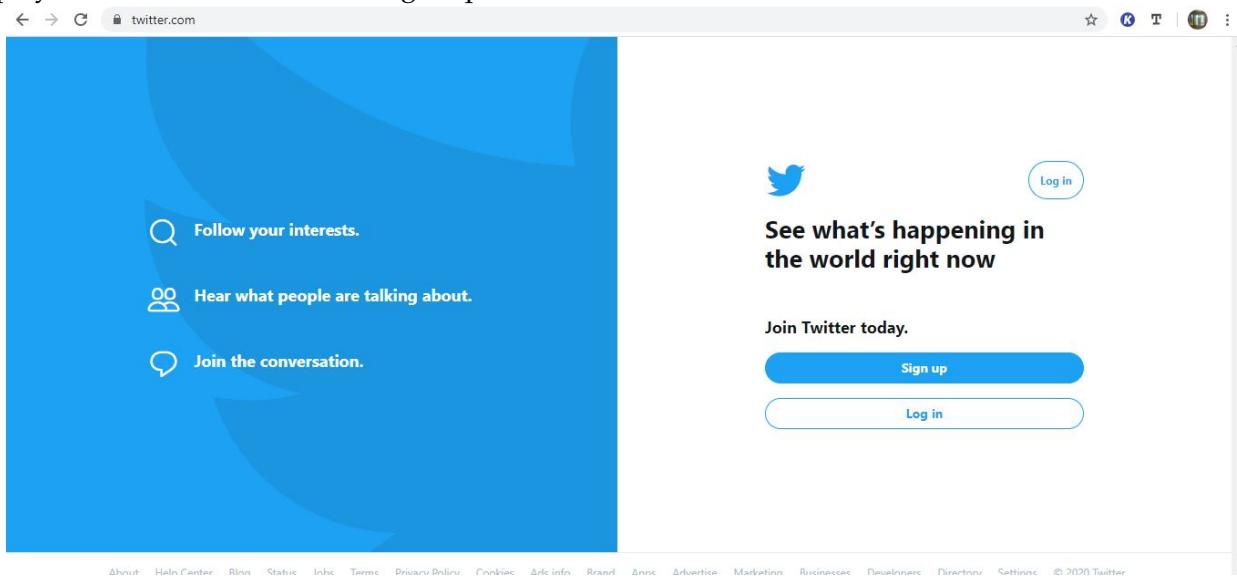
### 3 Installation

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

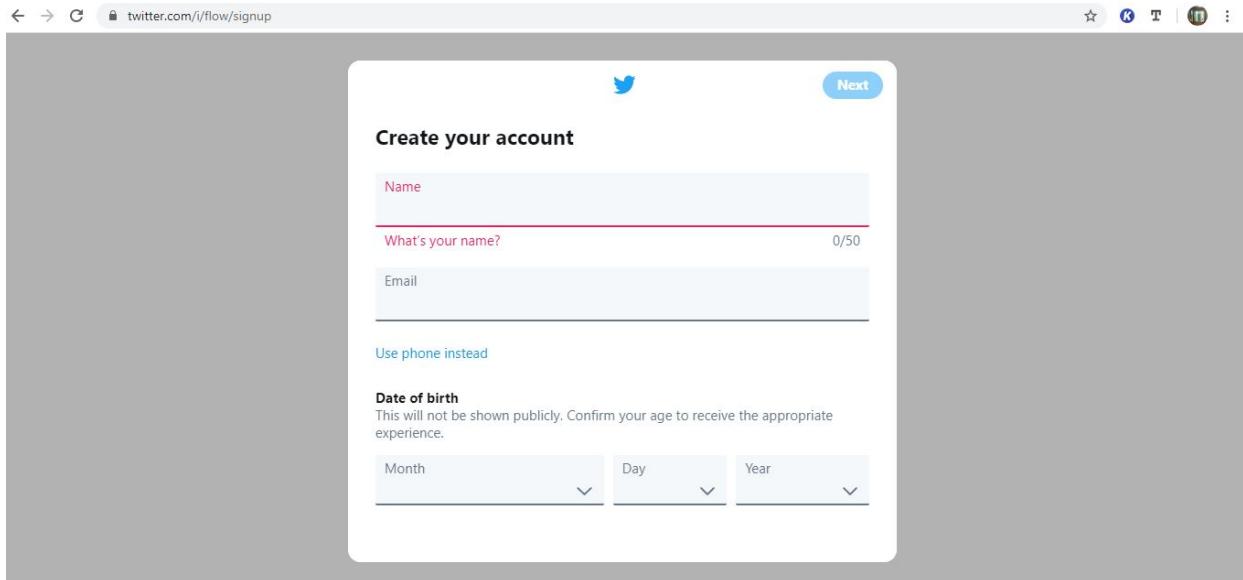
- Set up a Twitter user and developer account.
- Set up a Yelp! Account.
- Install a REST Client (We will use Insomnia)

#### 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":

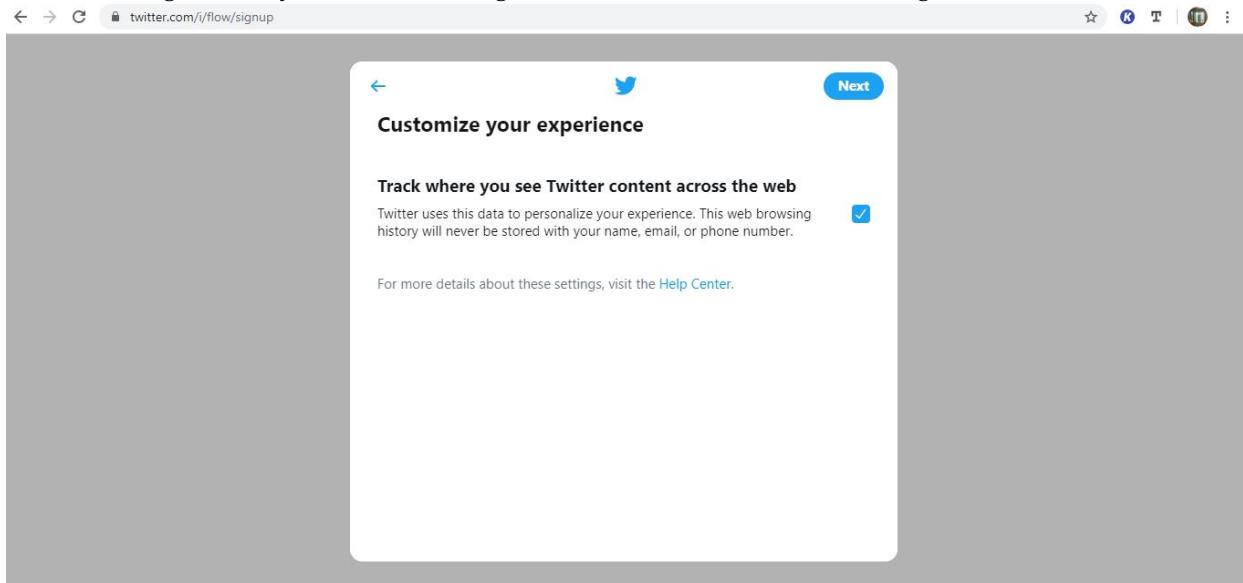


You can create an account using either your email or your phone. For developer accounts, **you will need both**, so ensure you have access to a phone number that allows for text message verification. You can start signing up by email by first clicking "Use email". Next, put in your birth date:



A screenshot of a Twitter account creation form. At the top right is a blue "Next" button. Below it is a section titled "Create your account". There are two input fields: "Name" and "What's your name?", with a character limit of 0/50. Below these is an "Email" field. A link "Use phone instead" is visible. Under "Date of birth", there is a note: "This will not be shown publicly. Confirm your age to receive the appropriate experience." Three dropdown menus for "Month", "Day", and "Year" are shown.

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



A screenshot of a Twitter customization form. At the top right is a blue "Next" button. Below it is a section titled "Customize your experience". There is a checkbox labeled "Track where you see Twitter content across the web", which is checked. A note below the checkbox states: "Twitter uses this data to personalize your experience. This web browsing history will never be stored with your name, email, or phone number." A link "For more details about these settings, visit the Help Center." is present.

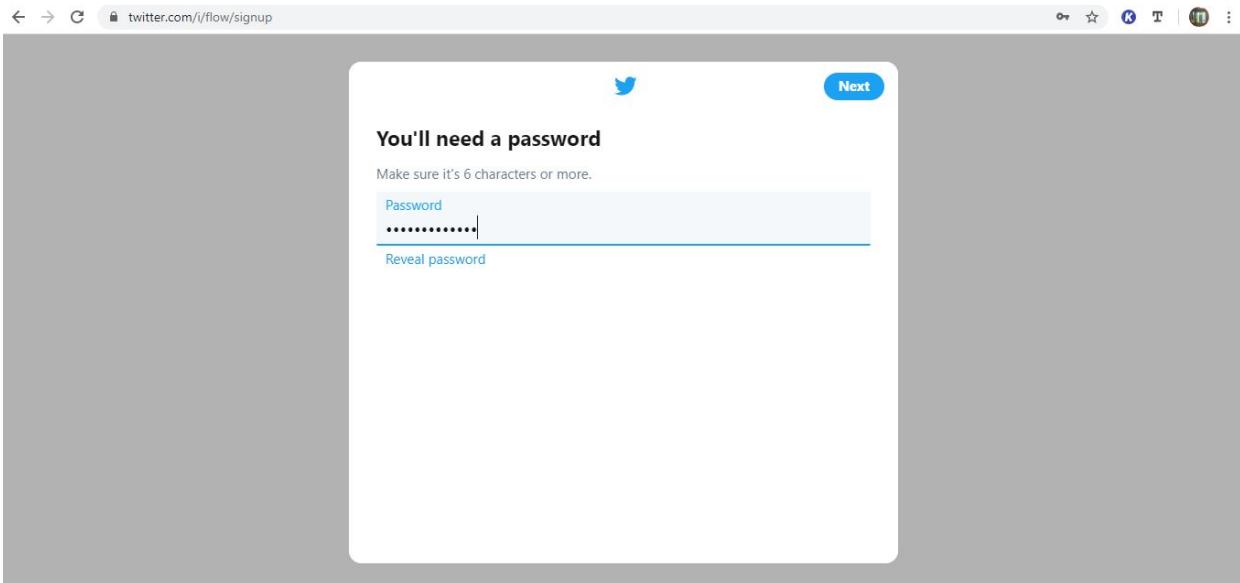
Now verify your details, and click "Sign Up":

The screenshot shows a web browser window for Twitter's account creation process at [twitter.com/i/flow/signup](https://twitter.com/i/flow/signup). The page is titled "Step 3 of 5" and asks to "Create your account". It contains three input fields: "Name" (Myles Garvey, Ph.D), "Email" (myles.garvey@rutgers.edu), and "Birth Date" (Mar 6, 1988). Below the inputs is a small text block about terms and conditions, followed by a blue "Sign up" button.

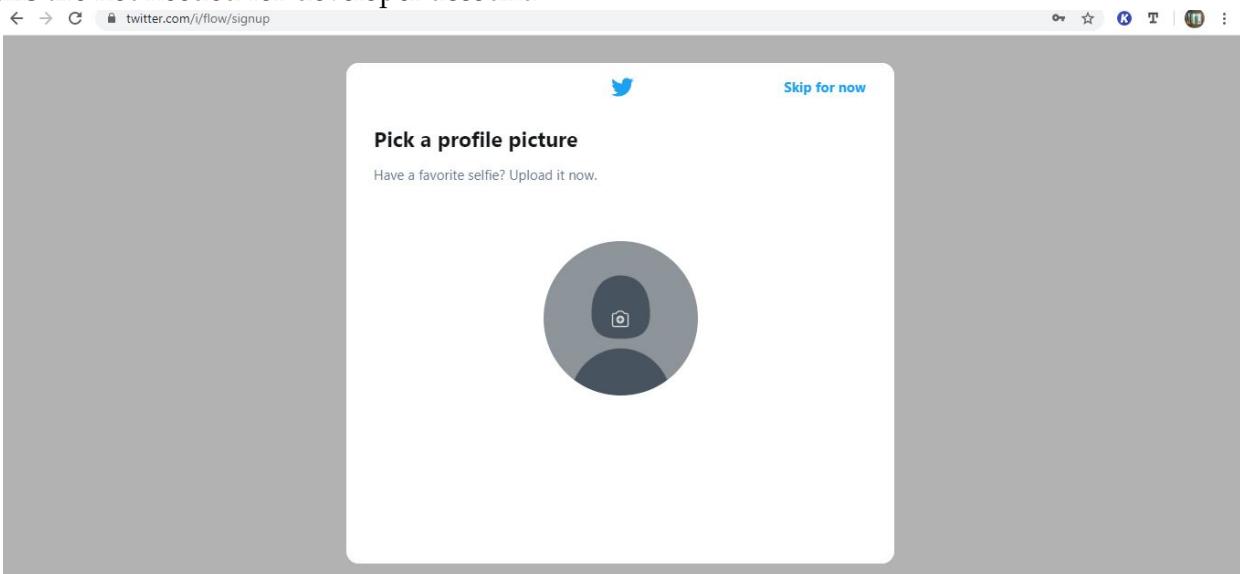
Twitter will now send a code to your email/phone (depending on how you signed up). Get the number and put it in the text box. After, click "Next":

The screenshot shows a web browser window for Twitter's account creation process at [twitter.com/i/flow/signup](https://twitter.com/i/flow/signup). The page is titled "We sent you a code" and instructs the user to enter it below to verify myles.garvey@rutgers.edu. It features a text input field labeled "Verification code" and a "Next" button in the top right corner. There is also a link "Didn't receive email?".

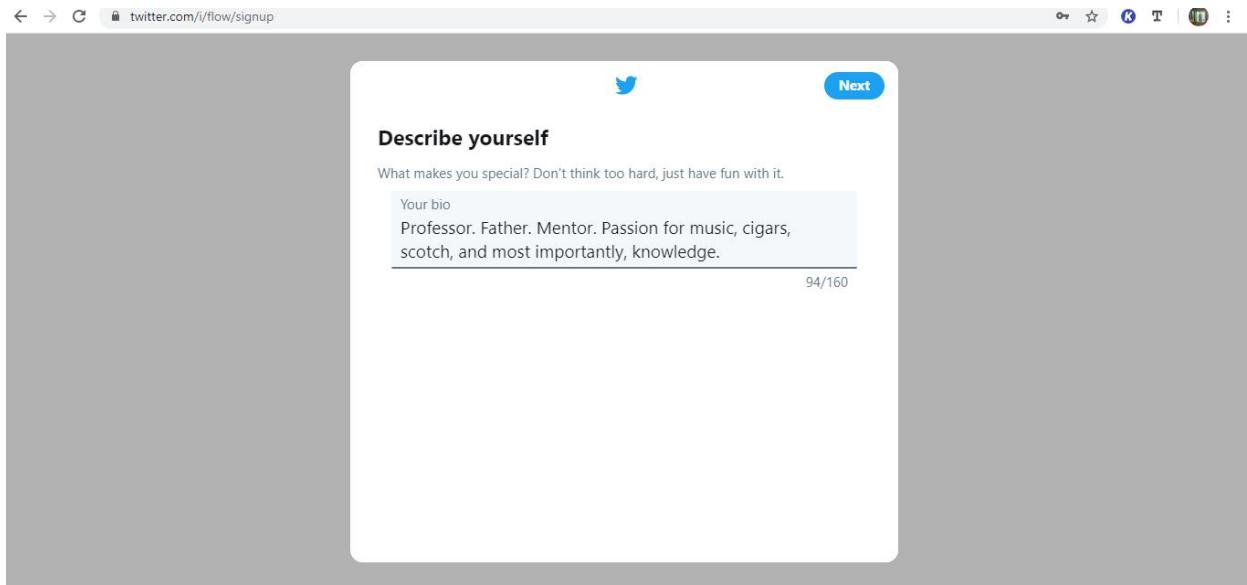
Now create a password and click "Next":



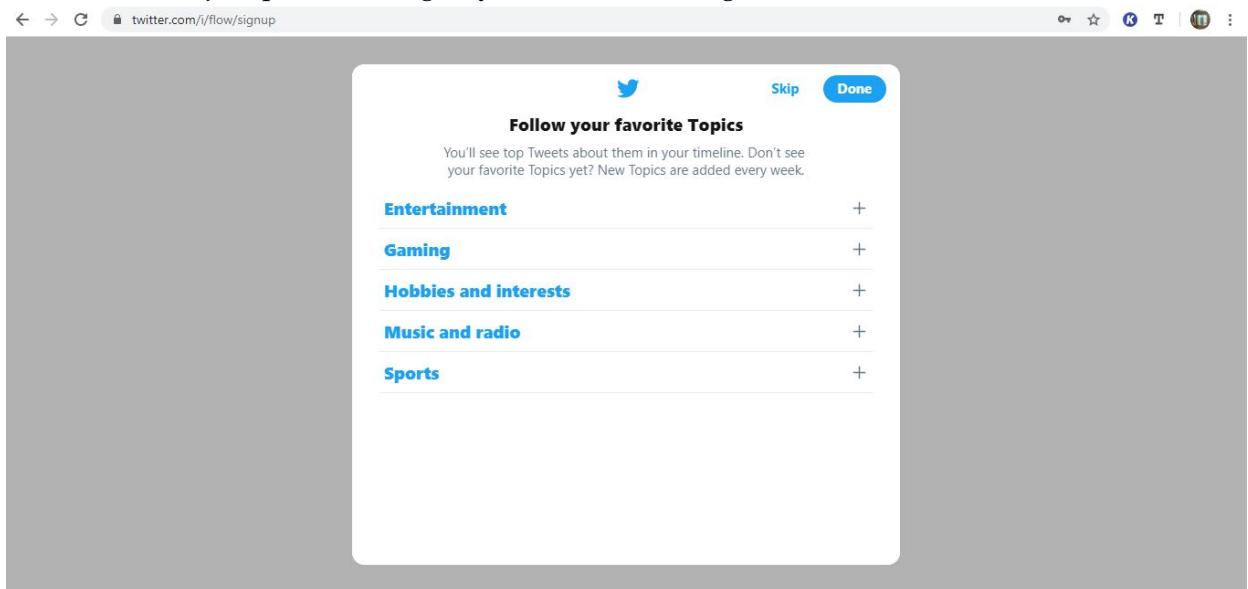
You don't need a profile picture if you do not want one. Click "Skip for now", the remaining details are not needed for developer account.



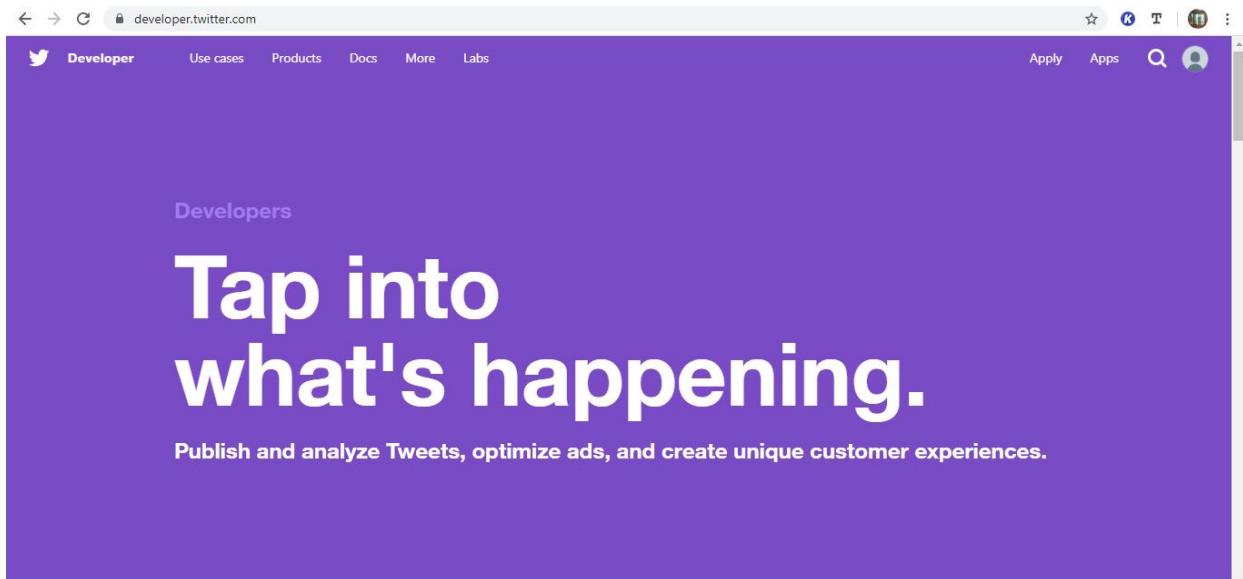
Now take a quick moment to describe yourself (this will be public info, just keep that in mind). I forget if Twitter allows you to keep it blank. If not, then just write nonsense.



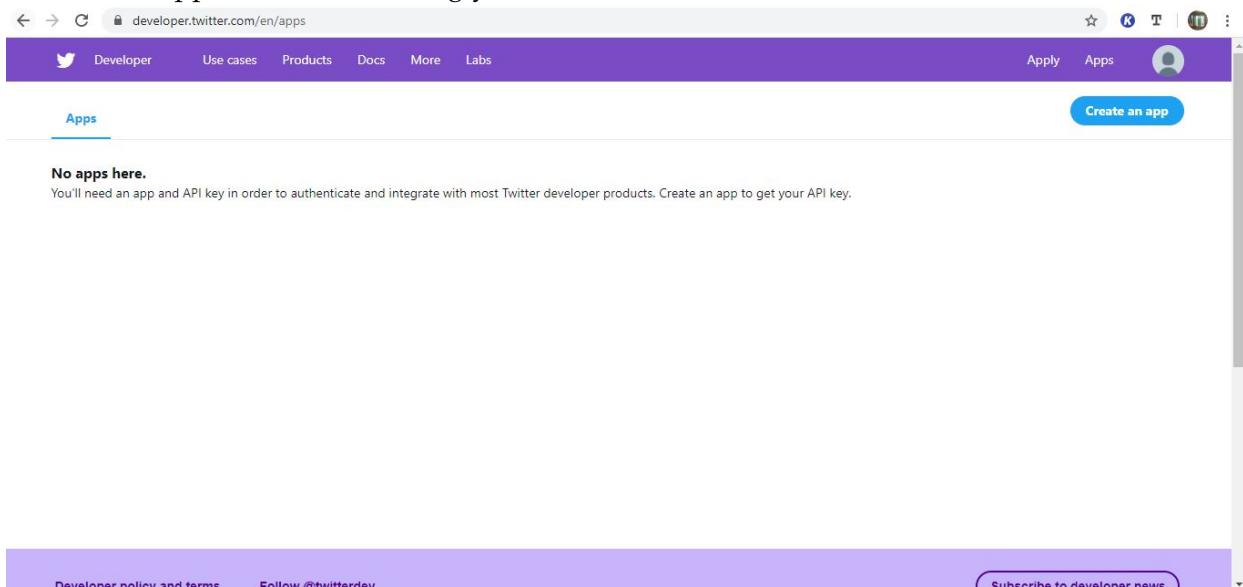
Now the rest is just personalising of your Twitter. Just ignore, and click "Done".



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:

The screenshot shows the Twitter Developer Apps page. At the top, there's a navigation bar with links for 'Developer', 'Use cases', 'Products', 'Docs', 'More', and 'Labs'. On the right side of the header, there are buttons for 'Apply', 'Apps', and a user profile icon. Below the header, a message says 'No apps here.' followed by a note: 'You'll need an app and API key in order to authenticate and integrate with most Twitter developer products. Create an app to get your API key.' A modal window titled 'Please apply for a Twitter developer account' is open in the center. It contains text about managing existing apps and applying for a developer account to use premium APIs. It also states that as a developer platform, they support the health of conversation on Twitter and have introduced requirements for developers. At the bottom of the modal is a blue 'Apply' button.

Click "Apply":

The screenshot shows the 'Get access to the Twitter API' application process. At the top, there's a navigation bar with links for 'Developer', 'Use cases', 'Products', 'Docs', 'More', and 'Labs'. On the right side of the header, there are buttons for 'Apply', 'Apps', and a user profile icon. The main content area has a purple sidebar on the left with a smiling emoji icon, the text '#welcome', and a message: 'We're excited you want to use Twitter APIs and data! As a developer platform, our first responsibility is to our users: to provide a place that supports the health of conversation on Twitter.' Below this, it lists 'This application process helps us to:' with two items: '1. Prevent abuse of the Twitter platform.' and '2. Better understand and serve our...'. The main content area has a heading 'What is your primary reason for using Twitter developer tools?' with a sub-note: 'We'll help you on your path to getting the most out of Twitter APIs and data.' It then lists four categories: 'Professional' (for commercial uses), 'Hobbyist' (for a personal project), 'Academic' (for education or research, which is highlighted with a green checkmark), and 'Other' (I don't fit any of those). Each category has a corresponding icon and a brief description. At the bottom right of the main content area is a blue 'Next' button.

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

The screenshot shows the Twitter developer login process at [developer.twitter.com/en/application/login?useCase=12](https://developer.twitter.com/en/application/login?useCase=12). The user is on the 'Intended use' step. A sidebar on the left says 'Individual developer account'. The main area asks 'This is you, right?' and shows a profile card for 'Myles Garvey, Ph.D' (@PhGarvey). A note says: 'This @username will be the login for your developer account. The phone number associated with this Twitter @username is not verified. You must add a valid phone number and verify it prior to applying for developer access.' A blue button says 'Add a valid phone number'.

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:

The screenshot shows the 'Review' step of the Twitter developer login process. A green banner at the top says 'Your phone number is now verified'. The main area shows the same profile card for 'Myles Garvey, Ph.D' (@PhGarvey) with the note: 'This @username will be the login for your developer account.'

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

The screenshot shows the Twitter Developer API application intent page. The URL is developer.twitter.com/en/application/intent. The top navigation bar includes links for Developer, Use cases, Products, Docs, More, Labs, Apply, Apps, and a user profile icon. The main content area has a purple sidebar on the left with sections for 'Key things to keep in mind' (mentioning developer policies and safe spaces), 'Restricted uses' (mentioning surveillance), and 'Automation' (warning about rules). The main panel is titled 'How will you use the Twitter API or Twitter data?' and contains a sub-section 'In your words' with a text input field containing: 'I am a professor using this for my BAN 7100 course (Data Mining and Data Warehousing). My students will subsequently send in requests. I will be teaching them how to use the Twitter API using Insomnia and subsequently R.' A note at the top right says 'All fields are required unless marked optional'. At the bottom are 'Back' and 'Next' buttons.

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:

The screenshot shows the 'The specifics' step of the Twitter Developer API application intent process. The URL is developer.twitter.com/en/application/intent. The top navigation bar and sidebar are identical to the previous screenshot. The main panel has a purple sidebar with sections for 'Automation' (warning about rules), 'Be thorough' (instructions to understand use case), and 'Thank you for your thoughtful answers.' The main panel is titled 'The specifics' and contains a question 'Please describe how you will analyze Twitter data including any analysis of Tweets or Twitter users.' Below it is a text input field with the placeholder: 'My students will learn how to use the streaming data, searching Tweets, and pulling Tweet information from company Twitter Pages. We will be analyzing the data using sentiment and context analysis!'. A note at the bottom says 'Response must be at least 100 characters'. A checked checkbox next to the question indicates 'Yes' to planning to analyze Twitter data. At the bottom are 'Back' and 'Next' buttons.

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

## Workshop 3: Application Programming Interfaces (APIs)

The screenshot shows the Twitter Developer API application process. The URL is developer.twitter.com/en/application/intent. The page title is "Get access to the Twitter API". The current step is "Intended use". The navigation bar includes "Developer", "Use cases", "Products", "Docs", "More", "Labs", "Apply", "Apps", and a user profile icon.

The main content area displays three questions with "No" selected:

- Will your app use Tweet, Retweet, like, follow, or Direct Message functionality? (No)
- Do you plan to display Tweets or aggregate data about Twitter content outside of Twitter? (No)
- Will your product, service or analysis make Twitter content or derived information available to a government entity? (No)  
In general, schools, colleges, and universities do **not** fall under this category.

At the bottom right are "Back" and "Next" buttons.

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

The screenshot shows the Twitter Developer API application process. The URL is developer.twitter.com/en/application/terms. The page title is "Get access to the Twitter API". The current step is "Terms". The navigation bar includes "Developer", "Use cases", "Products", "Docs", "More", "Labs", "Apply", "Apps", and a user profile icon.

The main content area displays the "Terms" section:

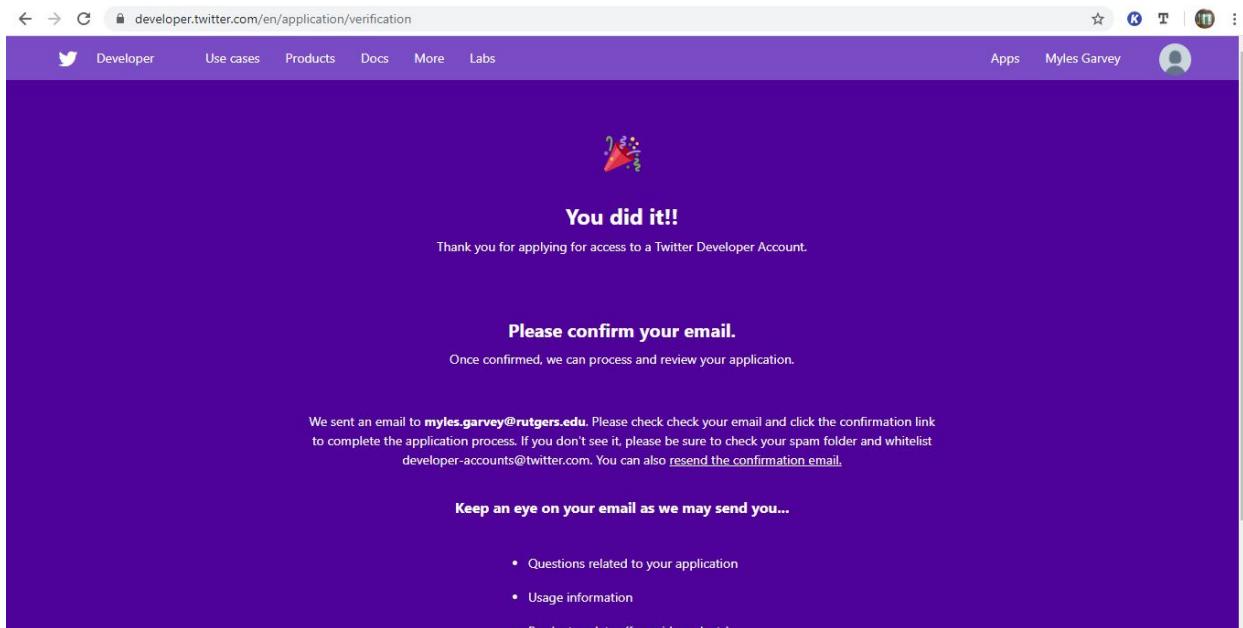
We know it's long. Thanks for taking the time to read our terms.

11. **Service** - Your websites, applications, hardware and other offerings that display or otherwise use Twitter Content.  
12. **User ID** - Unique identification numbers generated for each User that do not contain any personally identifiable information such as Twitter usernames or users' names.

A checkbox is checked, indicating agreement to the Developer Agreement and Twitter Developer Policy.

At the bottom right are "Back" and "Submit Application" buttons.

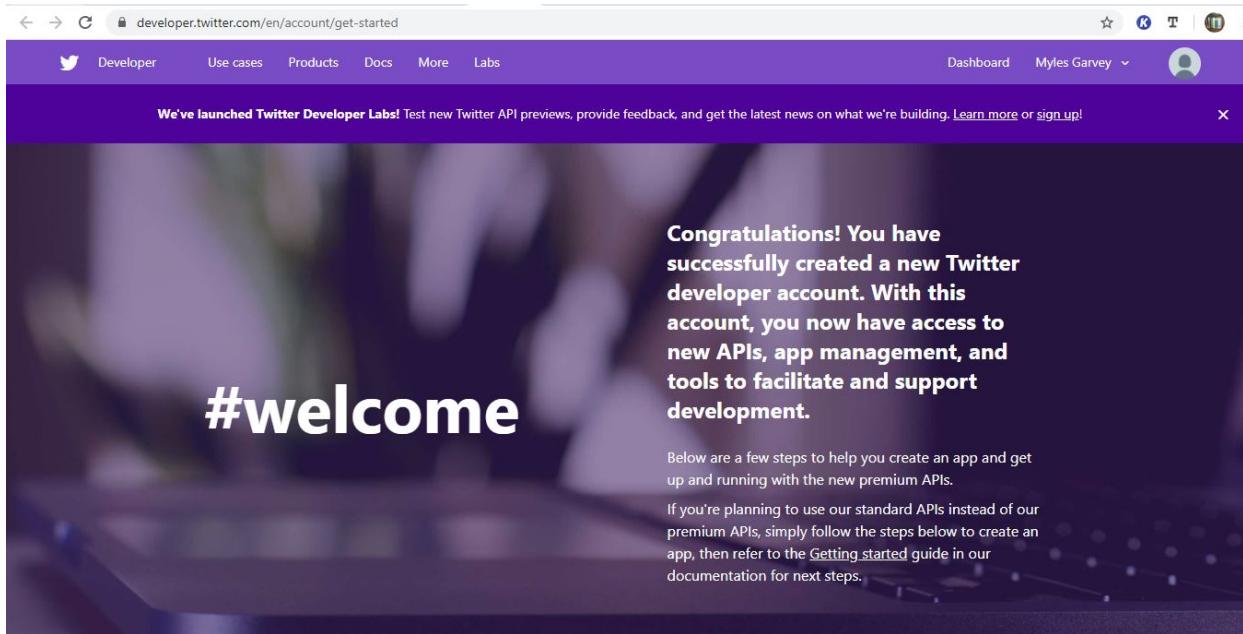
Read the terms of use and service, click the check box, and click "Submit Application":



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

A screenshot of an email inbox showing an incoming email from 'Twitter Developer Accounts &lt;developer-accounts@twitter.com&gt;' to 'Ph.D' at '0 minutes ago'. The subject is 'Email verification'. The email body starts with 'Hi Myles Garvey, Ph.D!' and thanks the recipient for applying. It asks them to confirm their email address to complete the application. A large orange button labeled 'Confirm your email' is prominently displayed. The message concludes with 'Thanks! The Twitter Dev team'.

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:

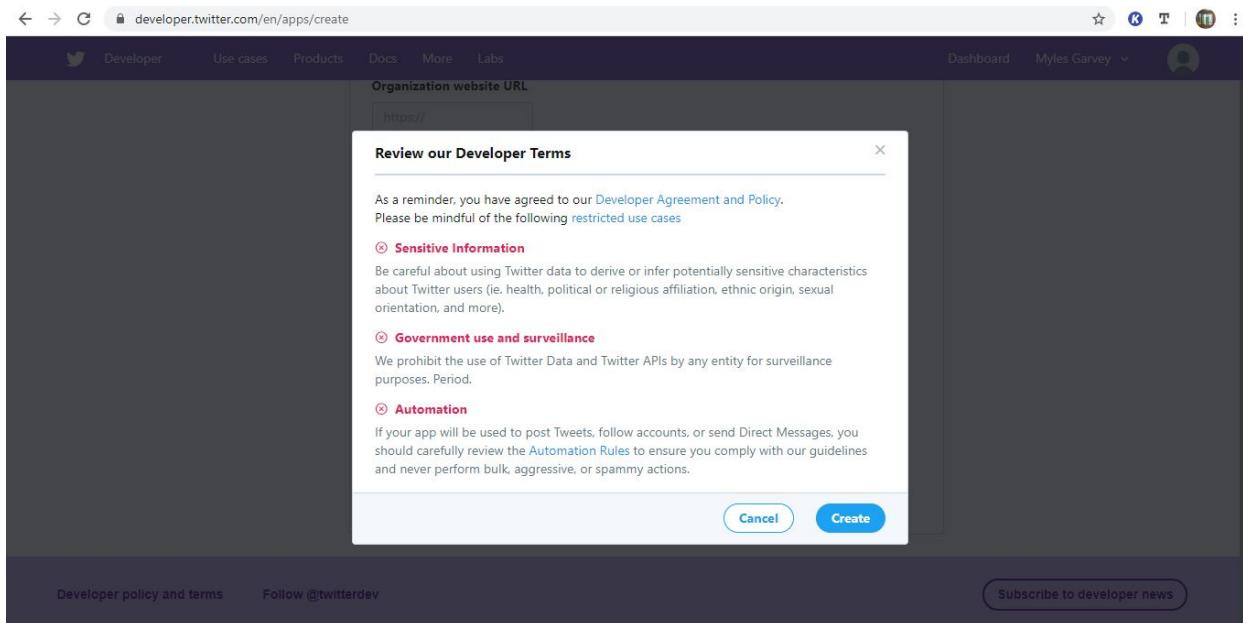
A screenshot of a web browser showing the Twitter Apps management page at developer.twitter.com/en/apps. The page has a purple header with navigation links like 'Developer', 'Use cases', 'Products', 'Docs', 'More', and 'Labs'. A purple bar at the top says 'We've launched Twitter Developer Labs! Test new Twitter API previews, provide feedback, and get the latest news on what we're building. Learn more or sign up!' Below this, a section titled 'Apps' shows a message: 'No apps here. You'll need an app and API key in order to authenticate and integrate with most Twitter developer products. Create an app to get your API key.' A blue button labeled 'Create an app' is visible.

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:

The screenshot shows the Twitter Developer App creation interface. On the left, there's a sidebar titled 'Understanding apps' with sections for 'What is an app?', 'Why register an app?', and 'Which products require an API key?'. The main area is titled 'App details' and contains fields for 'App name (required)' (set to 'BANDMDW') and 'Application description (required)' (containing the text 'This application will be a sandbox application for the students of WPU's BAN 7100 Data Mining Course.'). A note above the description says 'The following app details will be visible to app users and are required to generate the API keys needed to authenticate Twitter developer products.'

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:

This screenshot shows the continuation of the Twitter Developer App creation process. It includes fields for 'Organization website URL' (with 'https://' entered) and 'Tell us how this app will be used (required)'. The description in this field is: 'The application is only a sandbox application for my students to learn how to use the Twitter API and its functionality.' At the bottom are 'Cancel' and 'Create' buttons.



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:

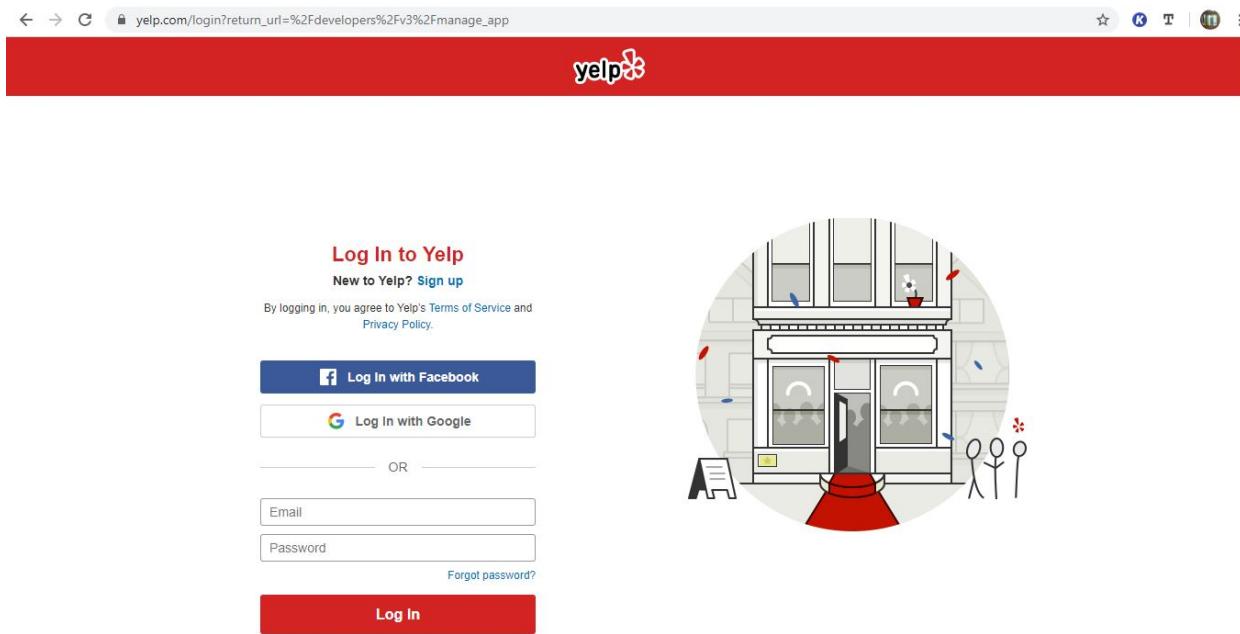
The screenshot shows the Twitter Developer dashboard with the app 'BANDMDW' selected. The 'Keys and tokens' tab is active. A yellow banner at the top states: 'Important notice about your access token and access token secret: To make your API integration more secure, we will no longer show your access token and access token secret beyond the first time that you generate it starting January 20, 2020. You will be able to regenerate it at anytime here, which will invalidate your current access token and secret. Please save this information if you need to access it. This does not affect your consumer API keys, which will still be shown here as they are below. To learn more, visit the Forums.' Below the banner, the 'Consumer API keys' section shows two keys: 'Ry1j8unRBomSm16PyDhcjvIc0' (API key) and 'Tn4OntfHhd53mzNRq2bPiwSCeRYhHQz7CneK9O4uBjsjwktN6k' (API secret key), each with a 'Regenerate' button. The 'Access token & access token secret' section shows 'None' and a 'Create' button.

## 3.2 Setting up a Yelp! Developer Account

Since we will also be exploring 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":

The screenshot shows the Yelp Fusion landing page. It features a large illustration of various electronic components like a water cooler, a monitor, and a keyboard. Below the illustration, the heading 'Discover the Power of Yelp Fusion' is displayed. A subtext reads: 'Harness the power of Yelp. Unlock a wealth of content and data from over 50 million businesses. See what you can do with the full suite of Yelp Fusion APIs.' A prominent red 'Get Started' button is centered. At the bottom, there are three sections: 'Documentation' (with a link to 'Learn more about Yelp Fusion and how to start building with our APIs.'), 'Community' (with a link to 'Check out the Yelp Fusion GitHub and provide feedback. Let's build together!'), and 'Request VIP Upgrade' (with a link to 'How might your project might benefit from more Yelp Fusion?').

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:

[Sign in with Google](#)

## Choose an account

to continue to [yelp.com](#)



Myles Garvey

[mylesgarvey@gmail.com](mailto:mylesgarvey@gmail.com)



Professor Garvey

[professorgarvey1988@gmail.com](mailto:professorgarvey1988@gmail.com)

Signed out

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:

General

**My App**

Manage App

Client ID  
is76gtMtk9uEN7dubb57vg

API Key  
rgPevl3YOGivKqD4p2PN3uydtZe9Bj4E35Uve-uv1a-0EHFP5y4\_h4yxwpnRhAenVbtQ1qi4h-gQKMNKwByv1nImejG6  
EPLRKFc-kTPKAqDkNDzyJ20I08CEYEUxhYX

App Name  
WPUAPP

App Website  
Optional

Industry  
School / Education

Company  
Optional

Contact Email  
professorgarvey1968@gmail.com

Fusion VIP

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:

To enable GraphQL, please join the Yelp Developer Beta Program.

## Intro to GraphQL

GraphQL is a query language for APIs. What does this mean for you? Unlike regular SOAP or REST APIs, GraphQL gives you the ultimate flexibility in being able to specify in your API requests specifically what data you need, and get back exactly that.

As a query language, it provides you with a lot of flexibility that most normal APIs will not. Without needing to recreate endpoints, you can provide developers with the same functionality as a bulk endpoint. Your queries will be cleaner and easier to understand by combining multiple queries into one request.

**What's the difference between Yelp's GraphQL API and the regular API?**

The regular API is very well structured and specifically defined. The endpoints have their set requests and responses and that's what you get whether or not that matches your usage pattern. GraphQL lets you control all of this so that the way you consume the data matches exactly what you need.

This is both a pro and a con. If your use case does not require all of the data, GraphQL can speed up your requests as we do less work on the server-side to fulfill those requests. Conversely, if you need all of the data in one request, your requests could slow down as we do more work to fulfill these requests.

The data itself returned from the regular API and GraphQL should be identical, with one caveat: we're only returning 1 photo from GraphQL at the moment compared to 3 from the regular API.

**Should I use GraphQL over the regular API?**

That's up to you! If you're someone who favors the ability to customize their requests to fit your specific use-case, GraphQL will be a good fit. GraphQL is still in beta period, which means that we can't promise that it's fully stable or that some of the existing implementation won't change. If you're not comfortable with finding a bug or two and making changes to your app to keep up with us, the regular API will be a better fit.

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):

The screenshot shows a web browser displaying the [Yelp GraphQL documentation](https://yelp.com/developers/graphql/guides/intro). The left sidebar has a navigation menu under the 'GraphQL' category, with 'Intro to GraphQL' highlighted. The main content area discusses how to use the API, mentioning the beta status and rate limits. It includes a code example using curl to make a POST request to the GraphQL endpoint, specifying the access token and content type, and providing a sample JSON response.

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:

```

1 $ curl -X POST -H "Authorization: Bearer ACCESS_TOKEN" -H "Content-Type:
  application/graphql" https://api.yelp.com/v3/graphq --data '
2 {
3   business(id: "garaje-san-francisco") {
4     name
5     id
6     alias
7     rating
8     url
9   }
10 }

```

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>:

The screenshot shows the Insomnia REST client homepage. At the top, there's a navigation bar with links for 'Docs', 'Pricing', 'Signup', and a 'Download' button. Below the header, a large purple banner features the text 'Debug APIs like a human, not a robot' and 'Finally, a REST client you'll *love*'. A 'Download for Windows' button is located at the bottom of the banner. The main content area is a dark-themed interface showing a POST request to 'POST /v0\_base\_url/tasks'. The JSON payload is as follows:

```

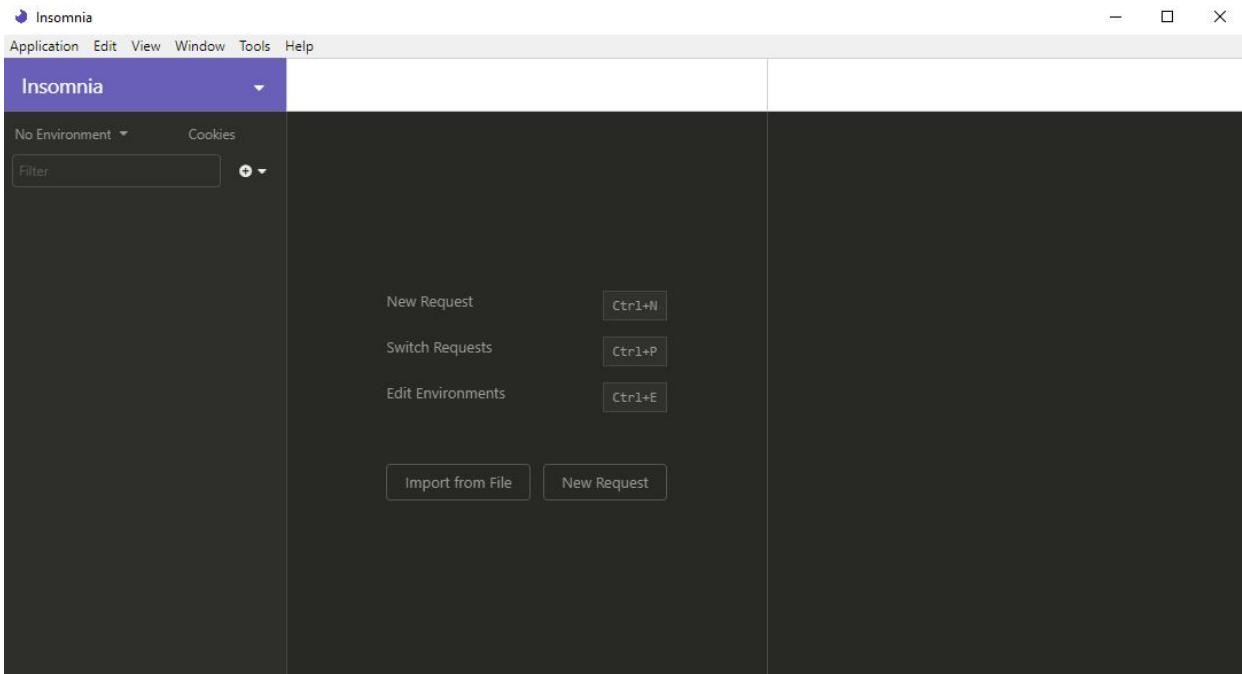
1+ {
2+   "name": "Download Insomnia",
3+   "due_date": "n",
4+   "duration_min": 300,
5+   "completed": false,
6+   "user": "users.admin",
7+   "languages": "languages.english_usa"
8+ }
  
```

The response shows a successful 200 OK status with a response body containing task details.

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:

The screenshot shows the 'Download Insomnia' page. It features a heading 'Download Insomnia' and a subtext 'So you can finally GET some REST 😊'. Below this are three download links: 'OS X 10.9+', 'Windows 7+', and 'Ubuntu 14.04+'. Each link is accompanied by its respective operating system logo (Apple, Windows, and Linux).

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:

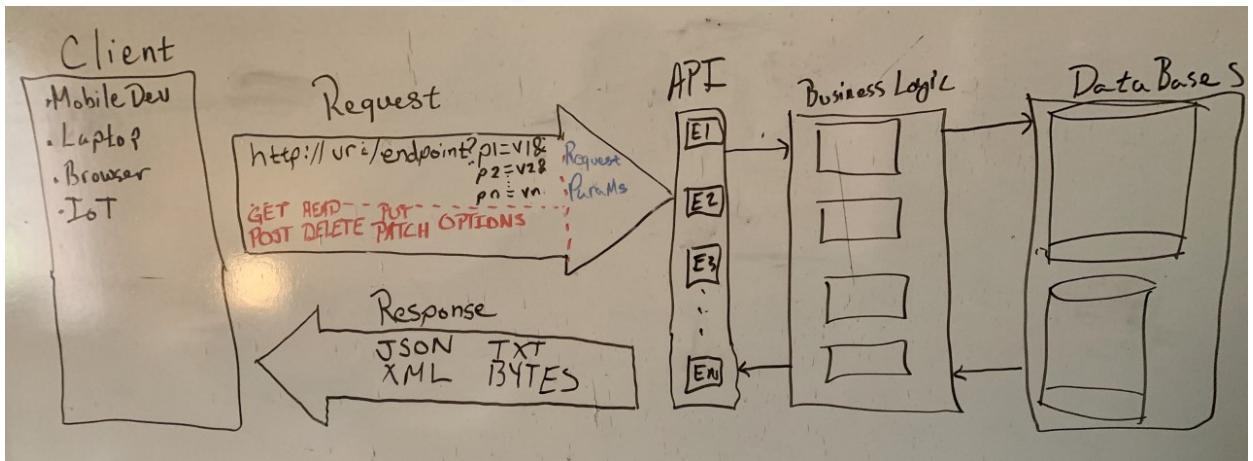


If you do not, look for the software on your computer, and try to load it. Congrats! We are now ready to start the workshop!

## 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\_timeline).
- Get a list of parameter names and values (in this instance, we have only a single input, namely screen\_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](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:

The screenshot shows a web browser displaying the Dog API documentation at <https://dog.ceo/dog-api/documentation/>. The page has a sidebar on the left with links for 'Documentation', 'Breeds list', 'About', and 'Submit your dog'. The main content area features a logo of a dog head with the letters 'API' inside. Below the logo, the text 'ENDPOINTS' is displayed, followed by five links: 'List all breeds', 'Random image', 'By breed', 'By sub-breed', and 'Browse breed list'. Under the 'List all breeds' link, the URL <https://dog.ceo/api/breeds/list/all> is shown. Below this, under the heading 'LIST ALL BREEDS', there is a JSON response example:

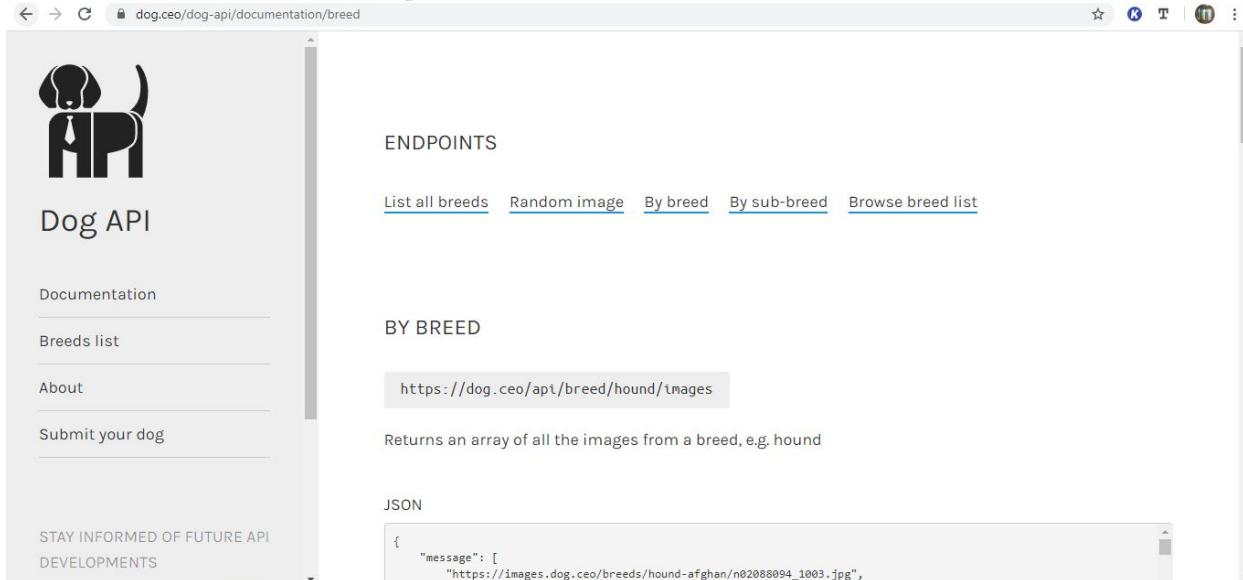
```
{
  "message": [
    "affenpinscher": [],
    "african": [],
    "airedale": []
}
```

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:



```
{
  "message": [
    "affenpinscher", ["african"], "airedale", [], "akita", [], "appenzeller", [], "australian", ["shepherd"], "baseji", [], "beagle", [], "bluetick", [], "borzoi", [], "bouvier", [], "boxer", [], "brabancon", [], "brisard", [], "buahund", ["norwegian"], "bulldog", ["boston", "english", "french"], "bulldog", ["staffordshire"], "cain", [], "cattledog", ["australian"], "chiwuhua", [], "chow", [], "clumber", [], "cockeroo", [], "collie", ["border"], "coonhound", ["american"], "corgi", ["cardigan"], "cotondetulear", [], "dachshund", [], "dalmatian", [], "dane", ["great"], "deerhound", ["scottish"], "dhole", [], "dingo", [], "doberman", [], "elkhound", ["norwegian"], "entlebucher", [], "eskimo", ["cardigan"], "frise", ["bichon"], "germanshepherd", [], "greyhound", ["italian"], "grounder", [], "hound", [], "afghan", "basset", "blood", "english", "ibizan", "walker", "husky", [], "keeshond", [], "kelpie", [], "komondor", [], "kuvasz", [], "labrador", [], "leonberg", ["german"], "lhasa", [], "malamute", [], "malinois", [], "maltese", [], "mastiff", ["bully"], "english", "tibetan", "mexicanhairless", [], "mari", [], "mountain", ["berneese"], "newfoundland", [], "otterhound", [], "papillon", [], "pekinse", [], "pembroke", [], "pinscher", ["miniature"], "pitbull", [], "pointer", ["german", "germanlonghaired"], "pomeranian", [], "poodle", ["miniture", "standard", "toy"], "pug", [], "puglie", [], "pyrenees", [], "redbone", [], "retriever", ["chesapeake", "curly"], "flatcoated", "golden", "ridgeback", ["rhodesian"], "rottweiler", [], "saluki", [], "samoyed", [], "schipperke", [], "schnauzer", ["giant", "miniature"], "setter", ["gordon", "irish"], "sheepdog", ["english", "shetland"], "shiba", [], "shihtzu", [], "spaniel", ["blenheim", "brittany", "cocker", "irish", "japanese", "sussex", "welsh"], "springer", ["english"], "stbernard", [], "terrier", ["american", "australian", "bedlington", "border", "dandie", "fox", "irish", "kerryblue", "lakeland", "norfolk", "norwich", "patterdale", "russell", "scottish", "sealyham", "silky", "tibetan", "toy", "westhighland", "wheaten", "yorkshire"], "vizsla", [], "waterdog", ["spanish"], "weimaraner", [], "whippet", [], "wolfhound", ["irish"], "status": "success"
  ]
}
```

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:



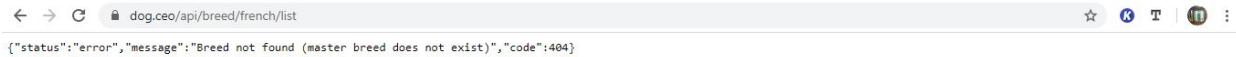
The screenshot shows the Dog API documentation page. On the left, there's a sidebar with links to Documentation, Breeds list, About, and Submit your dog. The main content area has a logo with a stylized dog head and the text "ENDPOINTS". Below this are several navigation links: List all breeds, Random image, By breed, By sub-breed, and Browse breed list. Under the "BY BREED" section, there's a URL: <https://dog.ceo/api/breed/hound/images>. A description below the URL states: "Returns an array of all the images from a breed, e.g. hound". Further down, there's a "JSON" section with a code snippet showing a partial JSON response:

```
{
  "message": [
    "https://images.dog.ceo/breeds/hound-afghan/n02088094_1003.jpg",
    ...
  ]
}
```

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](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).

The screenshot shows the homepage of the spoonacular.com/food-api. At the top, there's a navigation bar with icons for back, forward, search, and user profile. The URL is spoonacular.com/food-api. Below the navigation is the logo for "spoonacular API". To the right of the logo are links for "OVERVIEW", "DOCS", "PRICING", "TERMS", "APPLICATIONS", and a prominent green "START NOW" button. The main headline reads "The only food API you'll ever need." Below this, there's a section about the complex food ontology, mentioning years of engineering work. To the right of this text is a diagram illustrating the API's interconnected data model. It features three main circular nodes: a blue node labeled "menu items" containing a burger icon, a green node labeled "products" containing a shopping cart icon, and a red node labeled "menu items" containing a dollar sign icon. Small circular icons representing various food items like a pepper, a tomato, and a carrot are connected by lines to the central nodes, showing the relationships between ingredients, recipes, nutrition, and allergens.

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:

**Sign Up**

Email: garveym2@wpunj.edu

Password: .....|

**Sign Up**

You have an account already? [Log in here.](#)

Forgot your password? [Reset it here.](#)

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 lose it:

**Profile**

On this page you find your information and can sign up for developer news.

Email: garveym2@wpunj.edu

>Password: [Change Password](#)

API Key: 4e3b6e030be24f7a9bde5d9a2b877cda

Show / Hide API Key   Generate New API Key

[DELETE ACCOUNT](#)

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:

The screenshot shows the homepage of the Spoonacular API documentation at [spoonacular.com/food-api/docs](https://spoonacular.com/food-api/docs). The top navigation bar includes links for OVERVIEW, DOCS, PRICING, TERMS, APPLICATIONS, and a green button labeled "MY CONSOLE". The main content area is titled "Documentation". It features a search bar and a sidebar with sections for "Recipes" (Search Recipes, Search Recipes by Nutrients, Search Recipes by Ingredients, Search Recipes Complex, Get Recipe Information, Get Recipe Information Bulk, Get Similar Recipes, Get Random Recipes, Autocomplete Recipe Search) and "Guides" (Authentication, Quotas, Show Images, List of Ingredients, Write a Chatbot, Diets, Intolerances, Cuisines, Meal Types, Recipe Sorting Options, Tutorial with RapidAPI). The main content area also includes a "Search Recipes" section with a "GET https://api.spoonacular.com/recipes/search" endpoint, response headers (Content-Type: application/json), and parameters (Name: str, Type: string, Example: The (natural language) recipe search query).

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:

The screenshot shows the same homepage as before, but the "Guides" section is now expanded. It lists various guides: Authentication, Quotas, Show Images, List of Ingredients, Write a Chatbot, Diets, Intolerances, Cuisines, Meal Types, Recipe Sorting Options, and a "Tutorial with RapidAPI" link. Below this, the "Widgets" section is shown with links for Visualize Recipe Ingredients by ID, Visualize Recipe Equipment by ID, and Visualize Recipe Price.

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):

The screenshot shows the Spoonacular API documentation for the food-api. The left sidebar lists endpoints under the 'Recipes' category, including Search Recipes, Search Recipes by Nutrients, Search Recipes by Ingredients, Search Recipes Complex, Get Recipe Information, Get Recipe Information Bulk, Get Similar Recipes, Get Random Recipes, Autocomplete Recipe Search, Get Recipe Equipment by ID, Get Recipe Price Breakdown by ID, Get Recipe Ingredients by ID, and Get Recipe Nutrition Widget by. The main content area is titled 'Authentication' and contains instructions for using an API key, with a note about putting it in the request URL.

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 usage. Usually, an 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 means 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!):

The screenshot shows the Spoonacular API documentation for the food-api. The left sidebar lists the same endpoints as the previous screenshot. The main content area is titled 'Quotas' and explains that each API plan comes with a daily quota of points. It states that usually, every request costs 1 point and 0.01 points per result returned. It also provides information on how to check the API response headers for quota usage and when the quota resets at midnight UTC.

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 their 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 screenshot shows the Spoonacular API documentation. On the left, there's a sidebar with a search bar and sections for Recipes (Search Recipes, Search Recipes by Nutrients, Search Recipes by Ingredients, Search Recipes Complex), Get Recipe Information, Get Recipe Information Bulk, Get Similar Recipes, Get Random Recipes, and Autocomplete Recipe Search. The main content area is titled 'Get Ingredient Information' and describes how to use an ingredient ID to get information about an ingredient. It includes a 'GET' request example: `https://api.spoonacular.com/food/ingredients/{id}/information`. Below this, there are 'Headers' (Content-Type: application/json) and 'Parameters' (Name: id, Type: number, Example: 9266, Description: The ingredient id).

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:

```
1 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:

```

1 {"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:

```

1 {
2     "results": [
3         {
4             "id": 246916,
5             "title": "Bison Burger",
6             "readyInMinutes": 45,
7             "servings": 6,
8             "image": "Buffalo-Burger-246916.jpg",
9             "imageUrls": [
10                 "Buffalo-Burger-246916.jpg"
11             ]
12         },
13         {
14             "id": 245166,
15             "title": "Hawaiian Pork Burger",
16             "readyInMinutes": 40,
17             "servings": 4,
18             "image": "Hawaiian-Pork-Burger-245166.jpg",
19             "imageUrls": [
20                 "Hawaiian-Pork-Burger-245166.jpg"
21             ]
22         },
23         {
24             "id": 219957,
25             "title": "Carrot & sesame burgers",
26             "readyInMinutes": 50,
27             "servings": 6,
28             "image": "Carrot---sesame-burgers-219957.jpg",
29             "imageUrls": [
30                 "Carrot---sesame-burgers-219957.jpg"
31             ]
32         },
33     }

```

```

34     "id":607109,
35     "title":"Turkey Zucchini Burger with Garlic Mayo",
36     "readyInMinutes":45,
37     "servings":6,
38     "image":"Turkey-Zucchini-Burger-with-Garlic-Mayo-607109.jpg",
39     "imageUrls":[
40         "Turkey-Zucchini-Burger-with-Garlic-Mayo-607109.jpg"
41     ]
42 },
43 {
44     "id":219871,
45     "title":"Halloumi aubergine burgers with harissa relish",
46     "readyInMinutes":20,
47     "servings":4,
48     "image":"Halloumi-aubergine-burgers-with-harissa-relish-219871.jpg",
49     "imageUrls":[
50         "Halloumi-aubergine-burgers-with-harissa-relish-219871.jpg"
51     ]
52 }
53 ],
54 "baseUri":"https://spoonacular.com/recipeImages/",
55 "offset":0,
56 "number":5,
57 "totalResults":67,
58 "processingTimeMs":111,
59 "expires":1578751620313
60 }

```

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/246916/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

```
1 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:

```
1 {
2   "vegetarian":false,
3   "vegan":false,
4   "glutenFree":true,
5   "dairyFree":true,
6   "veryHealthy":false,
7   "cheap":false,
8   "veryPopular":true,
9   "sustainable":false,
10  "weightWatcherSmartPoints":12,
11  "gaps":"no",
12  "lowFodmap":false,
13  "ketogenic":false,
14  "whole30":false,
15  "sourceUrl":"https://www.simplyrecipes.com/recipes/buffalo_burger/",
16  "spoonacularSourceUrl":"https://spoonacular.com/bison-burger-246916",
17  "aggregateLikes":5345,
18  "spoonacularScore":93.0,
19  "healthScore":24.0,
20  "creditsText":"Simply Recipes",
21  "sourceName":"Simply Recipes",
22  "pricePerServing":349.0,
23  "extendedIngredients":[
24    {
25      "id":17330,
```

```

26     "aisle": "Meat",
27     "image": "fresh-ground-beef.jpg",
28     "consistency": "solid",
29     "name": "ground bison",
30     "original": "2 pounds ground bison (buffalo)",
31     "originalString": "2 pounds ground bison (buffalo)",
32     "originalName": "ground bison (buffalo)",
33     "amount": 2.0,
34     "unit": "pounds",
35     "meta": [
36       "(buffalo)"
37     ],
38     "metaInformation": [
39       "(buffalo)"
40     ],
41     "measures": {
42       "us": {
43         "amount": 2.0,
44         "unitShort": "lb",
45         "unitLong": "pounds"
46       },
47       "metric": {
48         "amount": 907.185,
49         "unitShort": "g",
50         "unitLong": "grams"
51       }
52     }
53   },
54   {
55     "id": 99226,
56     "aisle": "Produce;Spices and Seasonings",
57     "image": "fresh-sage.png",
58     "consistency": "solid",
59     "name": "fresh sage",
60     "original": "2 tablespoons finely chopped fresh sage",
61     "originalString": "2 tablespoons finely chopped fresh sage",
62     "originalName": "finely chopped fresh sage",
63     "amount": 2.0,
64     "unit": "tablespoons",
65     "meta": [
66       "fresh",
67       "finely chopped"
68     ],
69     "metaInformation": [
70       "fresh",
71       "finely chopped"
72     ],
73     "measures": {
74       "us": {
75         "amount": 2.0,
76         "unitShort": "Tbsp",
77         "unitLong": "Tbsp"
78       },
79       "metric": {
80         "amount": 2.0,
81         "unitShort": "Tbsp",

```

```

82         "unitLong":"Tbsps"
83     }
84   }
85 },
86 {
87   "id":2047,
88   "aisle":"Spices and Seasonings",
89   "image":"salt.jpg",
90   "consistency":"solid",
91   "name":"salt",
92   "original":"1 1/2 teaspoons salt",
93   "originalString":"1 1/2 teaspoons salt",
94   "originalName":"salt",
95   "amount":1.5,
96   "unit":"teaspoons",
97   "meta":[
98
99 ],
100  "metaInformation":[
101
102 ],
103  "measures":{
104    "us":{
105      "amount":1.5,
106      "unitShort":"tsp",
107      "unitLong":"teaspoons"
108    },
109    "metric":{
110      "amount":1.5,
111      "unitShort":"tsp",
112      "unitLong":"teaspoons"
113    }
114  },
115 },
116 {
117   "id":1002030,
118   "aisle":"Spices and Seasonings",
119   "image":"pepper.jpg",
120   "consistency":"solid",
121   "name":"black pepper",
122   "original":"2 teaspoons black pepper",
123   "originalString":"2 teaspoons black pepper",
124   "originalName":"black pepper",
125   "amount":2.0,
126   "unit":"teaspoons",
127   "meta":[
128     "black"
129   ],
130   "metaInformation":[
131     "black"
132   ],
133   "measures":{
134     "us":{
135       "amount":2.0,
136       "unitShort":"tsp",
137       "unitLong":"teaspoons"

```

```

138     },
139     "metric": {
140       "amount": 2.0,
141       "unitShort": "tsp",
142       "unitLong": "teaspoons"
143     }
144   },
145   {
146     "id": 11282,
147     "aisle": "Produce",
148     "image": "brown-onion.png",
149     "consistency": "solid",
150     "name": "onion",
151     "original": "1/2 onion, finely chopped",
152     "originalString": "1/2 onion, finely chopped",
153     "originalName": "onion, finely chopped",
154     "amount": 0.5,
155     "unit": "",
156     "meta": [
157       "finely chopped"
158     ],
159     "metaInformation": [
160       "finely chopped"
161     ],
162     "measures": {
163       "us": {
164         "amount": 0.5,
165         "unitShort": "",
166         "unitLong": ""
167       },
168       "metric": {
169         "amount": 0.5,
170         "unitShort": "",
171         "unitLong": ""
172       }
173     }
174   },
175   {
176     "id": 1034053,
177     "aisle": "Oil, Vinegar, Salad Dressing",
178     "image": "olive-oil.jpg",
179     "consistency": "liquid",
180     "name": "extra virgin olive oil",
181     "original": "2 tablespoons extra virgin olive oil",
182     "originalString": "2 tablespoons extra virgin olive oil",
183     "originalName": "extra virgin olive oil",
184     "amount": 2.0,
185     "unit": "tablespoons",
186     "meta": [
187
188     ],
189     "metaInformation": [
190
191     ],
192     "measures": {
193

```

```

194     "us": {
195         "amount": 2.0,
196         "unitShort": "Tbsps",
197         "unitLong": "Tbsps"
198     },
199     "metric": {
200         "amount": 2.0,
201         "unitShort": "Tbsps",
202         "unitLong": "Tbsps"
203     }
204 },
205 },
206 {
207     "id": 6150,
208     "aisle": "Condiments",
209     "image": "barbecue-sauce.jpg",
210     "consistency": "solid",
211     "name": "barbecue sauce",
212     "original": "Smoky barbecue sauce",
213     "originalString": "Smoky barbecue sauce",
214     "originalName": "Smoky barbecue sauce",
215     "amount": 1.0,
216     "unit": "serving",
217     "meta": [
218
219     ],
220     "metaInformation": [
221
222     ],
223     "measures": {
224         "us": {
225             "amount": 1.0,
226             "unitShort": "serving",
227             "unitLong": "serving"
228         },
229         "metric": {
230             "amount": 1.0,
231             "unitShort": "serving",
232             "unitLong": "serving"
233         }
234     }
235 },
236 ],
237     "id": 246916,
238     "title": "Bison Burger",
239     "readyInMinutes": 45,
240     "servings": 6,
241     "image": "https://spoonacular.com/recipeImages/246916-556x370.jpg",
242     "imageType": "jpg",
243     "cuisines": [
244         "American"
245     ],
246     "dishTypes": [
247         "lunch",
248         "main course",
249         "main dish",

```

```

250     "dinner"
251   ],
252   "diets": [
253     "gluten free",
254     "dairy free"
255   ],
256   "occasions": [
257   ],
258   "winePairing": {
259     "pairedWines": [
260       "malbec",
261       "merlot",
262       "zinfandel"
263     ],
264     "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.",
265     "productMatches": [
266       {
267         "id": 438817,
268         "title": "Terrazas de los Andes Reserva Malbec",
269         "description": "Bright red color with purple shades. Intense floral and fruity notes. Presence of violets, ripe black cherry and plum aromas. Its 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 Mediterranean Ratatouille and Moist Semisweet Chocolate Cake over Mascarpone and Red Fruits Ice Cream.",
270         "price": "$21.99",
271         "imageUrl": "https://spoonacular.com/productImages/438817-312x231.jpg",
272         "averageRating": 0.8400000000000001,
273         "ratingCount": 5.0,
274         "score": 0.7775000000000001,
275         "link": "https://click.linksynergy.com/deeplink?id=QCiIS6t4gA&mid=2025
276 &murl=https%3A%2F%2Fwww.wine.com%2Fproduct%2Fterrazas-de-los-andes-reserva-
277 malbec-2014%2F162923"
278       }
279     ]
280   },
281   "instructions": "Cook the onions: Saut the onions in the olive oil over medium
282 -high heat until translucent. Turn off the heat and let it cool.\n\nMix onions,
283 sage, salt, pepper into ground bison meat: When the onions are cool enough to
284 touch, use your (clean) hands to gently mix them in with the bison burger meat,
285 and add everything else.\nDo not overwork the meat, it will result in a tough
286 burger. Just gently fold it until the onions, sage, salt and pepper are well
287 mixed in.\n\nForm patties: Form patties with the meat, using about 1/4 to 1/3
288 of a pound of meat per patty.\nHere's a tip on making the patty: if you press a
289 slight indentation in the center of each patty it will help keep the burgers
290 in a nice disk shape when cooking. Otherwise the burger will start to get a
291 little egg-shaped as the edges contract from cooking.\n\nGrill or fry the
292 burgers: Grill or fry the burgers on medium heat, about 6-7 minutes per side,
293 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
.\nDon't 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.",

281 "analyzedInstructions": [
282   {
283     "name": "",
284     "steps": [
285       {
286         "number": 1,
287         "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.",
288         "ingredients": [
289           {
290             "id": 4053,
291             "name": "olive oil",
292             "image": "olive-oil.jpg"
293           },
294           {
295             "id": 11282,
296             "name": "onion",
297             "image": "brown-onion.png"
298           }
299         ],
300         "equipment": [
301           ]
302       },
303     {
304       "number": 2,
305       "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.",
306         "ingredients": [
307           {
308             "id": 17330,
309             "name": "ground bison",
310             "image": "fresh-ground-beef.jpg"
311           },
312           {
313             "id": 11282,
314             "name": "onion",
315             "image": "brown-onion.png"
316           },
317           {
318             "id": 1002030,
319             "name": "pepper",
320             "image": "pepper.jpg"
321           },
322           {
323             "id": 17330,
324             "name": "bison",
325           }

```

```

326         "image": "fresh-ground-beef.jpg"
327     },
328     {
329         "id": 99226,
330         "name": "sage",
331         "image": "fresh-sage.png"
332     },
333     {
334         "id": 2047,
335         "name": "salt",
336         "image": "salt.jpg"
337     }
338 ],
339     "equipment": [
340     ]
341 },
342 {
343     "number": 3,
344     "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."
345     ,
346     "ingredients": [
347     {
348         "id": 1102047,
349         "name": "salt and pepper",
350         "image": "salt-and-pepper.jpg"
351     },
352     {
353         "id": 11282,
354         "name": "onion",
355         "image": "brown-onion.png"
356     },
357     {
358         "id": 99226,
359         "name": "sage",
360         "image": "fresh-sage.png"
361     }
362     ],
363     "equipment": [
364     ]
365 },
366 {
367     "number": 4,
368     "step": "Form patties: Form patties with the meat, using about 1/4  

to 1/3 of a pound of meat per patty.",
369     "ingredients": [
370     ],
371     "equipment": [
372     ]
373 },
374 {
375     "number": 5,
376     "step": "Cook the patties: Heat a large skillet over medium heat. Add oil to the
377     "
378 }
```

```

379     "step":"Here's a tip on making the patty: if you press a slight
380     indentation in the center of each patty it will help keep the burgers in a nice
381     disk shape when cooking. Otherwise the burger will start to get a little egg-
382     shaped as the edges contract from cooking.",
383     "ingredients": [
384     ],
385     "equipment": [
386     ],
387     {
388         "number":6,
389         "step":"Grill or fry the burgers: Grill or fry the burgers on
390         medium heat, about 6-7 minutes per side, less or more depending on the
391         thickness of the burger and the heat of the pan/grill, or until the internal
392         temperature is 140 F for medium rare, or 160 F for well done.",
393         "ingredients": [
394             ],
395             "equipment": [
396                 {
397                     "id":404706,
398                     "name":"grill",
399                     "image":"grill.jpg"
400                 },
401                 {
402                     "id":404645,
403                     "name":"frying pan",
404                     "image":"pan.png"
405                 }
406             ],
407             "length": {
408                 "number":7,
409                 "unit":"minutes"
410             }
411         },
412         {
413             "number":7,
414             "step":"A note on internal temperature. If you are getting the
415             ground meat from a source you trust (we got ours from Whole Foods) that does
416             their own grinding on site, or you grind your own meat, you can safely cook the
417             burgers rare or medium rare. Otherwise you'll want to cook the burgers until
418             well done.",
419             "ingredients": [
420                 ],
421                 "equipment": [
422                     ],
423                     {
424                         "number":8,
425                         "step":"Don't press on your burgers while cooking, and keep the
426                         flipping to a minimum.",
427                         "ingredients": [

```

```

424     ],
425     "equipment": [
426
427         ],
428     },
429     {
430
431         "number":9,
432         "step":"Let the burgers rest about 5 minutes before serving.",
433         "ingredients": [
434
435             ],
436             "equipment": [
437
438                 ],
439                 "length": {
440                     "number":5,
441                     "unit":"minutes"
442                 }
443             },
444             {
445
446                 "number":10,
447                 "step":"Serve the burger with lettuce and tomato, topped with a
smoky barbecue sauce.",
448                 "ingredients": [
449
450                     {
451                         "id":6150,
452                         "name":"barbecue sauce",
453                         "image":"barbecue-sauce.jpg"
454                     }
455                 ],
456                 "equipment": [
457
458                     ]
459                 }
460             ]
461 }

```

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

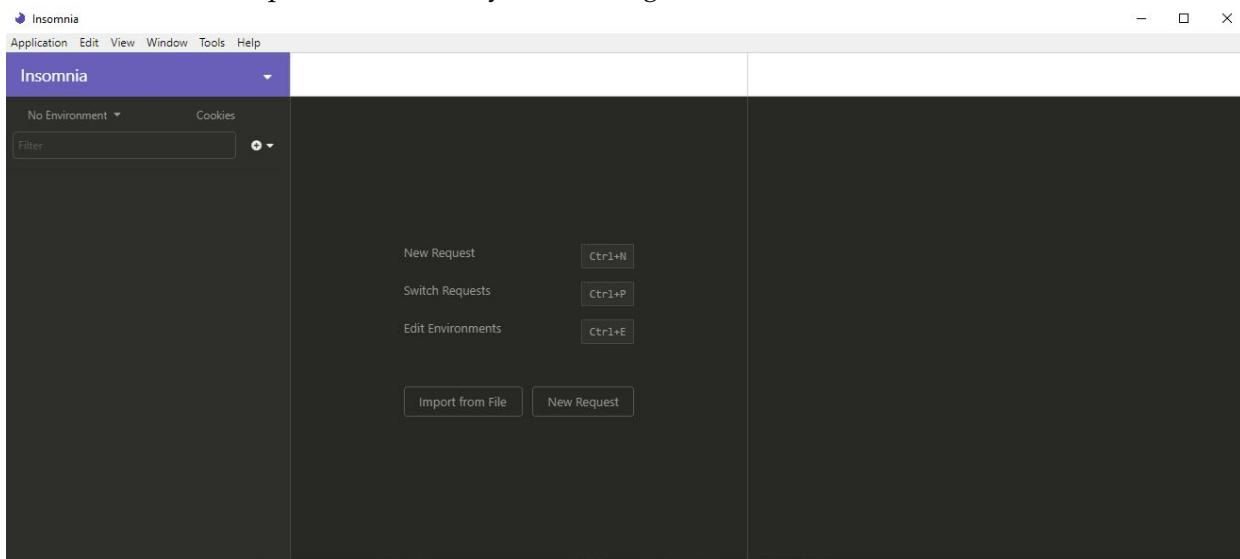
3. Go to <http://newsapi.org>. Look up the documentation. How does one authenticate themselves with this particular API? What do they need to do?

- 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.

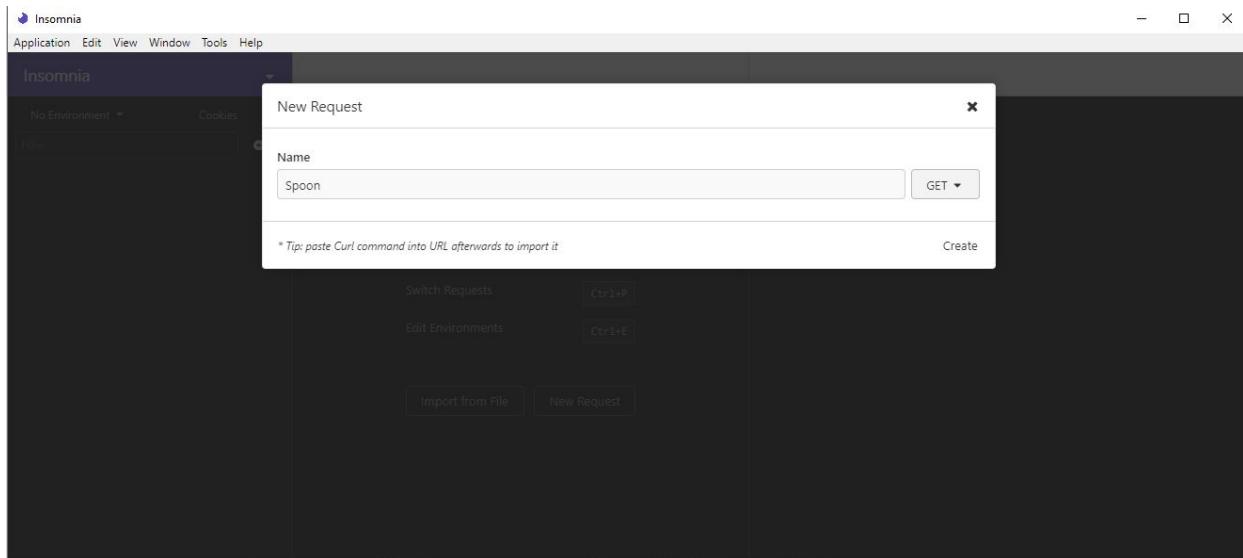
## 6 Tutorial 3: Gathering Data with Yellow Pages and Twitter in R

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.

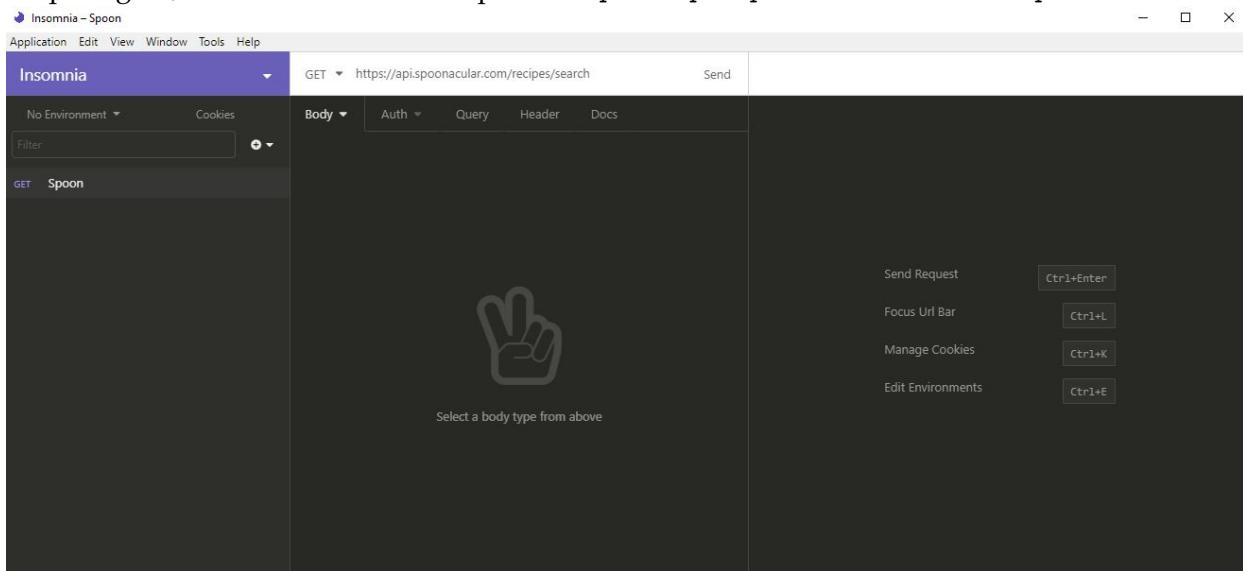
To illustrate how to send requests outside of the web browser, we will again use the spoonacular API. However, this time around, we will use Insomnia. This software allows us to easily construct requests, as well as returns responses and auto-formats them. This is not a tool to be used in production 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:



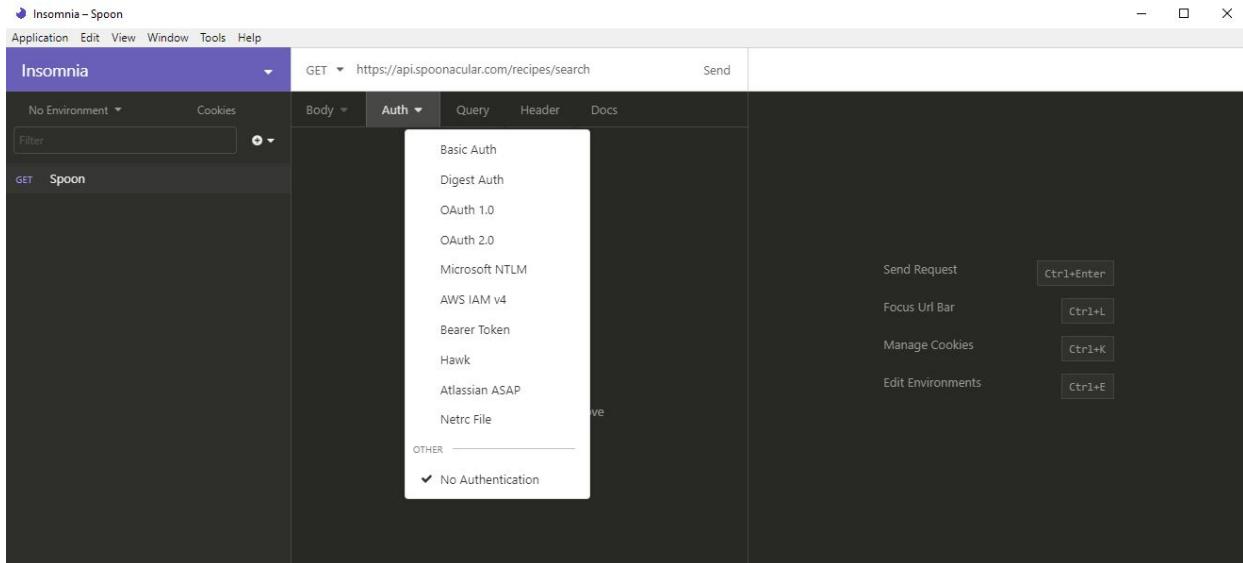
We are going to create a new request. Click on "New Request", which should bring up the screen below. Give the request a name (any name will suffice), and ensure it is marked as "GET" on the right hand side of the drop down. Why GET? Because the spoonacular documentation tells us that all the requests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

Parameter	Value
apiKey	4e3b6e030be24f7a9bde5d9a2b
query	steak
cuisine	italian

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:

```

1: {
2:   "results": [
3:     {
4:       "id": 879048,
5:       "title": "Grilled Steak Panzanella Salad with Tomato Vinaigrette",
6:       "readyInMinutes": 105,
7:       "servings": 4,
8:       "image": "grilled-steak-panzanella-salad-with-tomato-vinaigrette-
879048.jpg",
9:       "imageUrls": [
10:         "grilled-steak-panzanella-salad-with-tomato-vinaigrette-879048.jpg"
11:       ],
12:     },
13:     {
14:       "id": 498712,
15:       "title": "Philly Cheese Steak Pizza",
16:       "readyInMinutes": 45,
17:       "servings": 8,
18:       "image": "Philly-Cheese-Steak-Pizza-498712.jpg",
19:       "imageUrls": [
20:         "Philly-Cheese-Steak-Pizza-498712.jpg"
21:       ],
22:     },
23:     {
24:       "id": 556120,
25:       "title": "Steak and Blue Cheese Pizza",
26:       "readyInMinutes": 30,
27:       "servings": 4,
28:       "image": "Steak-and-Blue-Cheese-Pizza-556120.jpg"
29:     }
30:   ],
31:   "store.books[*].author"
32: }

```

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:

```

$ curl -X POST -H "Authorization: Bearer ACCESS_TOKEN" -H "Content-Type: application/graphql"
{
  business(id: "tnhfDv5iL8EaGSXZgiuQGg") {
    name
    id
    alias
    rating
    url
  }
}

```

Make sure to copy the single quotes too!

You get the response:

```

{
  "data": {
    "business": {
      "name": "Garaje",
      "id": "tnhfDv5iL8EaGSXZgiuQGg",
      "alias": "garaje-san-francisco",
      "rating": 4.5,
      "url": "https://yelp.com/biz/garaje-san-francisco"
    }
  }
}

```

Your response should look exactly like what you passed in! To make it easier for you to experiment with different queries we've set up [GraphiQL](#) an interactive web console. We've also provided many interactive examples all

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:

The screenshot shows the Yelp Fusion API documentation page. At the top, there's a navigation bar with links for 'Fusion API', 'GraphQL', and 'Manage App'. Below the navigation, there are two main sections: 'General' and 'Yelp Fusion'. Under 'General', there's a 'Search' section with a placeholder code snippet for a GraphQL query:

```
{
  search(argument_1: "value_1",
         argument_2: "value_2"
       ...) {
    field_1
    field_2 {
      nested_field_1
      ...
    }
  }
}
```

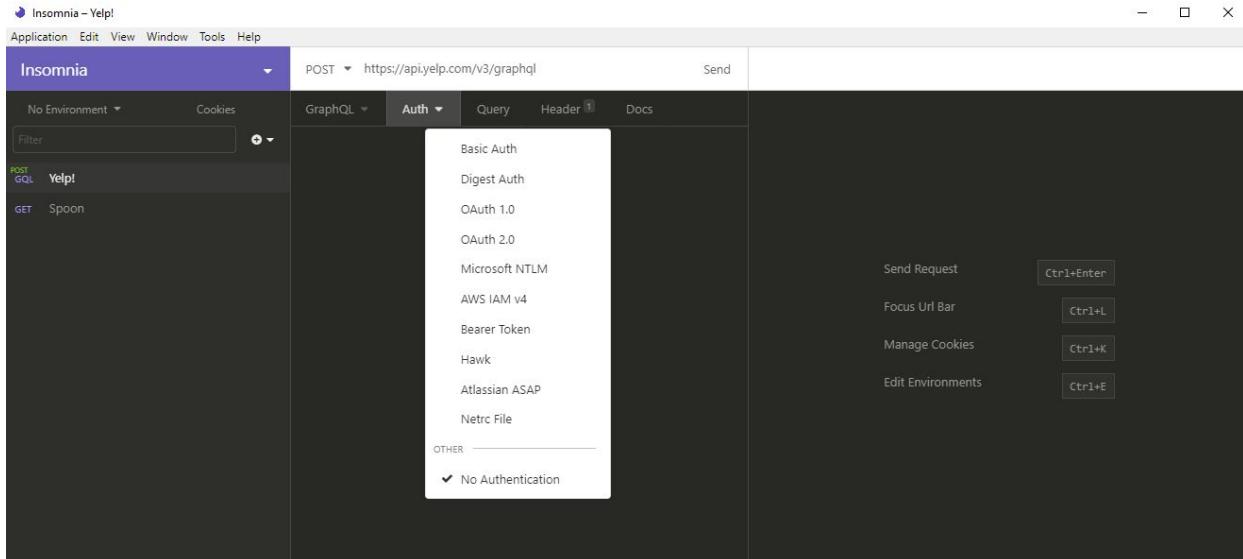
Under 'Yelp Fusion', there's a 'Sample query:' section with another GraphQL query:

```
{
  search(term: "burrito",
         location: "san francisco",
         limit: 5) {
    total
    business {
      name
      url
    }
  }
}
```

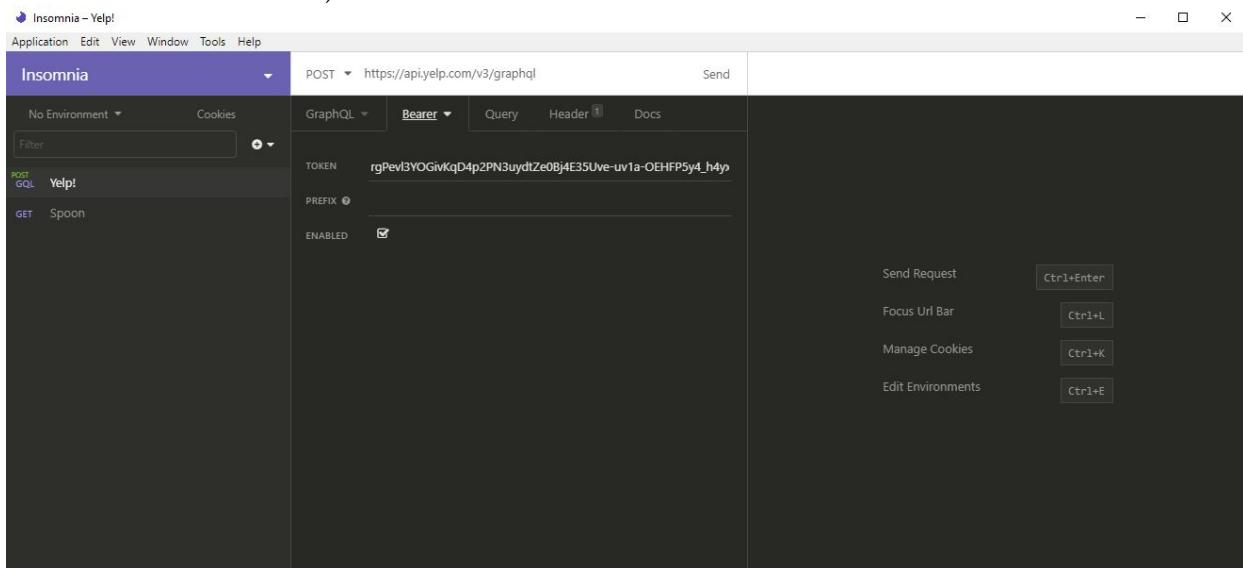
Now, let's use the GraphQL to make requests (by the way, other APIs also use a GraphQL. Whichever you decide to use is your choice). Open Insomnia and create a new request. This time, ensure it is a POST request, and that the body type is GraphQL:

The screenshot shows the Insomnia application window. A 'New Request' dialog box is open in the foreground. The 'Name' field contains 'Yelp!', the 'Method' dropdown is set to 'POST', and the 'Body Type' dropdown is set to 'GraphQL'. The main Insomnia interface shows a toolbar with 'No Environment' and 'Cookies' buttons, and a bottom row with 'Import from File' and 'New Request' buttons.

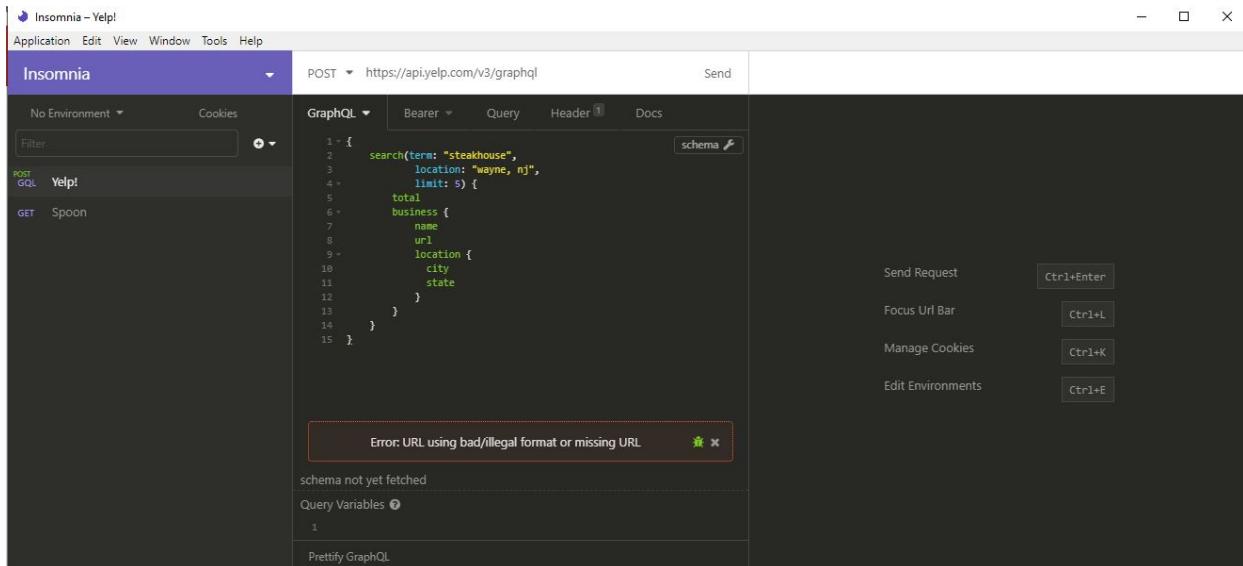
Add in the endpoint to the GraphQL, which for Yelp! is <https://api.yelp.com/v3/graphq1>. Also, click the drop down box for "Auth", and click on "Bearer Token":



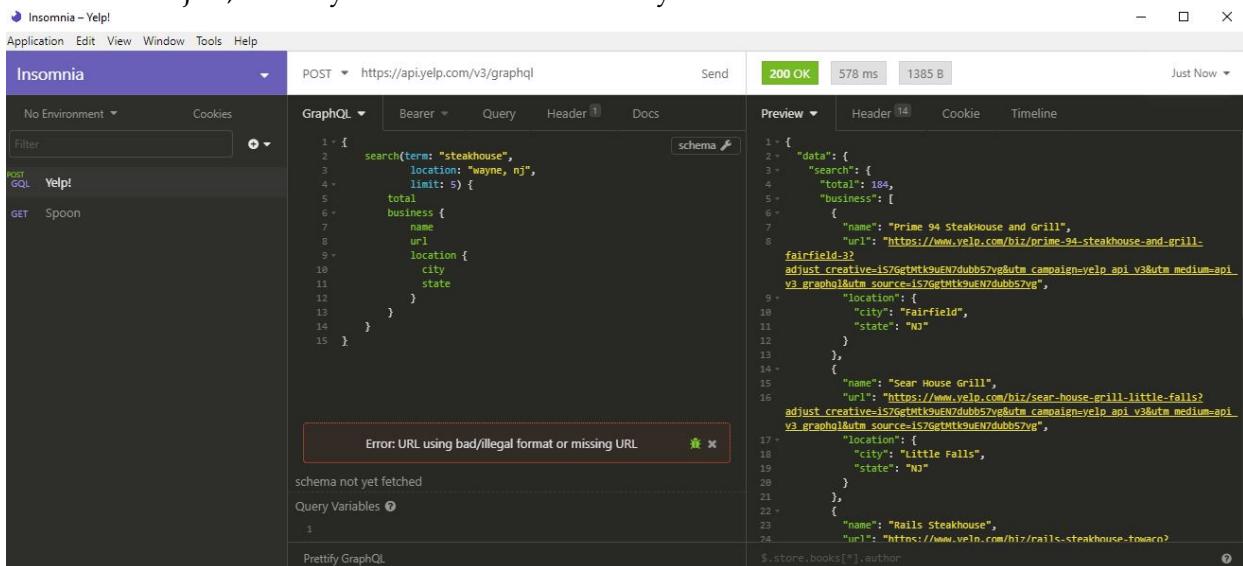
Now, input the token id from your Yelp! "Manage App" screen (see earlier in the tutorial, I TOLD YOU TO SAVE THIS!):



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



Finally, we have our output shown. 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:

The screenshot shows the Twitter Developer API Keys and Tokens page. At the top, there is a yellow banner with the text: "Important notice about your access token and access token secret. To make your API integration more secure, we will no longer show your access token and access token secret beyond the first time that you generate it starting **January 21, 2020**. You will be able to regenerate it at anytime here, which will invalidate your current access token and secret. Please save this information if you need to access it. This does not affect your consumer API keys, which will still be shown here as they are below. To learn more, [visit the Forums](#)."

**Keys and tokens**  
Keys, secret keys and access tokens management.

**Consumer API keys**  
Ry1J8unRBomSm16PyDhcjvlc0 (API key)  
Tn4OmtfHhd53mzNRq2bPiwSceRYhHQz7CneK9O4uBjsjwktN6k (API secret key)

[Regenerate](#)

**Access token & access token secret**  
121430877323366400-aokPY8aPbVjxf29fO7FktuceDOPHd3 (Access token)  
w7jyjBdsID1O2paypCCivRGdkl4wD4J134f4Jzb0ZqpGe (Access token secret)  
Read and write (Access level)

[Revoke](#) [Regenerate](#)

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

The screenshot shows the Insomnia REST Client interface. The top bar shows "Insomnia - Twitter" with menu options: Application, Edit, View, Window, Tools, Help. The main area shows a GET request to "https://api.myproduct.com/v1/users". The left sidebar lists API endpoints: GET Twitter, GET GQL, GET Yelp!, and GET Spoon. The right sidebar contains keyboard shortcuts: Send Request (Ctrl+Enter), Focus Url Bar (Ctrl+L), Manage Cookies (Ctrl+K), and Edit Environments (Ctrl+E). A central message says "Select an auth type from above" with a lock icon.

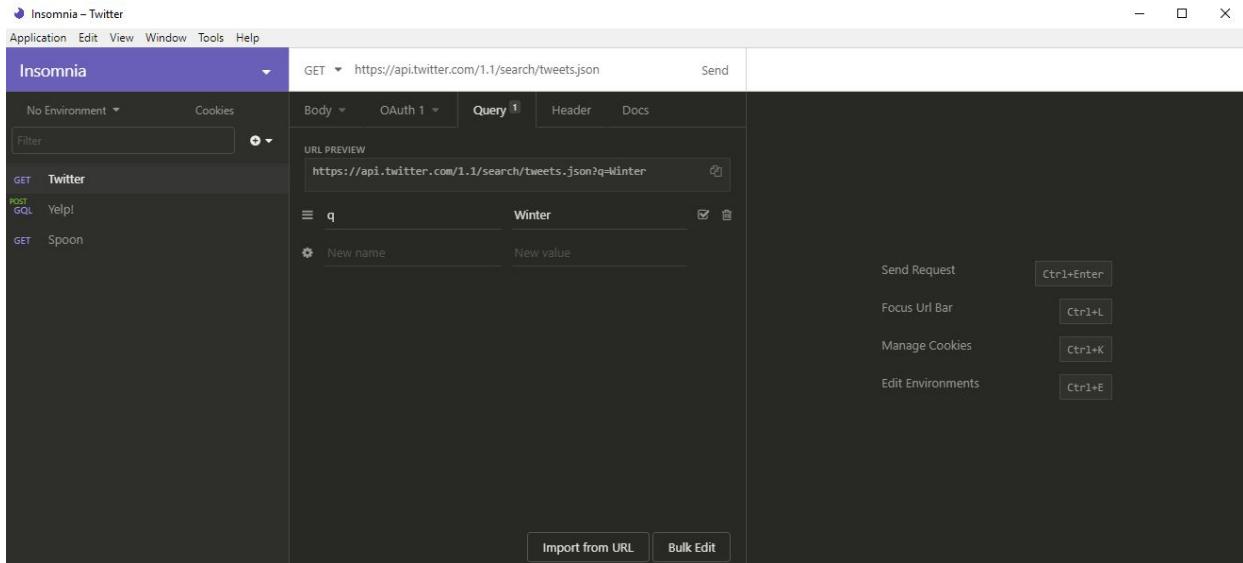
We will be using the Search Tweets endpoint in the Twitter API. You can find the documentation for it here:

The screenshot shows the Twitter Developer API documentation for the Search Tweets endpoint. The URL in the browser is [developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets](https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets). The page has a purple header with links for Developer, Use cases, Products, Docs, More, and Labs. On the right, it shows Dashboard and Myles Garvey. A search bar at the top says "Search all documentation...". The main title is "Search Tweets". Below it, there's a "Basics" section with a "Tweets" subsection containing links like "Post, retrieve and engage with Tweets" and "Get Tweet timelines". To the right, there's a sidebar with "API reference contents ^" and links for "Standard search API", "Enterprise search APIs", and "Premium search APIs". The main content area is titled "Standard search API" and describes how to return a collection of relevant Tweets matching a specified query. It notes that the service is not exhaustive and provides links for search operators and timeline best practices.

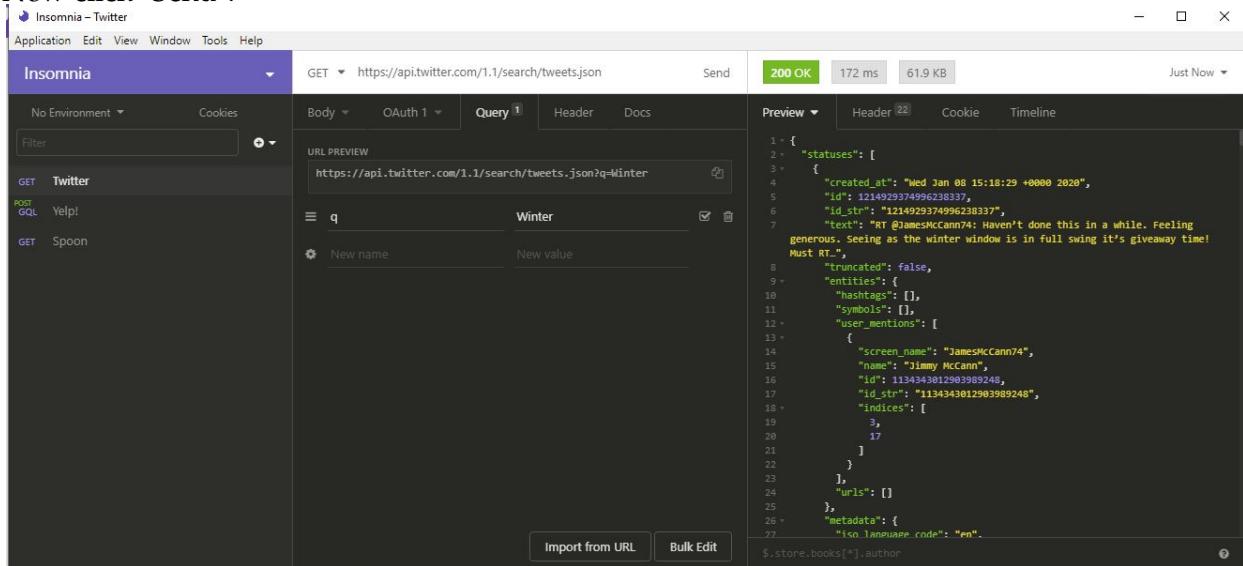
In Insomnia, click on OAuth 1 for the authentication method. Using the keys and tokens from earlier, put all 4 into the 4 respective boxes in this screen:

The screenshot shows the Insomnia REST Client interface. The left sidebar lists environments: "Insomnia" (selected), "No Environment", and "Cookies". Under "Insomnia", there are requests for "Twitter" (GET, q="Yelp!"), "Spoon" (GET), and "GQL" (POST). The main panel shows a request for "https://api.twitter.com/1.1/search/tweets.json" using the "OAuth 1" method. The OAuth parameters are filled with values from earlier: CONSUMER KEY (Ry1J8unRBomSm16PyDhcjvld0), CONSUMER SECRET (Tn40ntlhhd53mzNRq2bPiwSCeRYhHQz7CneK9O4uBjsjw), TOKEN (121430877332336640-aokPV8aPbVjxf29fO7FktuceDOPH), and TOKEN SECRET (w7jyjd5iD1O2paypCCivRGdkI4wD4J134F4Jzb0ZqpGe). Other fields include SIGNATURE METHOD (HMAC-SHA1), CALLBACK URL (empty), VERSION (1.0), TIMESTAMP (empty), REALM (empty), NONCE (empty), VERIFIER (empty), and ENABLED (checked). On the right, there are buttons for "Send Request" (Ctrl+Enter), "Focus Url Bar" (Ctrl+L), "Manage Cookies" (Ctrl+K), and "Edit Environments" (Ctrl+E).

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):

```

1 > library(httr)
2 >
3 > #Create OAuth Token (Customer Key and Secret)
4 > app <- oauth_app("twitter",
5 +                         key="Ry1J8unRBomSm16PyDhcjvlc0",
6 +                         secret = "Tn40ntfHhd53mzNRq2bPiwSCeRYhHQz7CneK904uBJsjwktn6k"
7 + )
8 >
9 > #Create OAuth Credentials (Access Key and Secret)
10 > credentials <- list(oauth_token = "1214308773323366400-
11 +                         aokPY8aPbVjxf29f07FktucedOPHd3",
12 +                         oauth_token_secret = "
13 +                         w7jyjBd5iD102paypCCivRGdkI4wD4J134F4Jzb0ZqpGe")
14 >
15 > #Ensure that we put this in the header of the request
16 > params <- list(as_header = TRUE)
17 >
18 > #Create the token with the twitter endpoints
19 > token <- Token1.0$new(app = app,
20 +                         endpoint = oauth_endpoints("twitter"),
21 +                         params = params,
22 +                         credentials = credentials,
23 +                         cache = FALSE)
24 > #Endpoint URL:
25 > eURL<-"https://api.twitter.com/1.1/search/tweets.json?q=Winter"
```

```

26 >
27 > #Convert any http request errors into R errors
28 > stop_for_status(req)
29 >
30 > #Grab the Response Data. R will return it as a list rather than in JSON
31 > response<-content(req)
32 >
33 > #See the structure of the response:
34 > names(response)
35 [1] "statuses"           "search_metadata"
36 >
37 > #We only care about the Twitter statuses:
38 > statuses<-response$statuses
39 >
40 > #Take A Peek Inside Some of The Properties of the First Tweet:
41 > names(statuses[[1]])
42 [1] "created_at"          "id"
43 [3] "id_str"               "text"
44 [5] "truncated"             "entities"
45 [7] "extended_entities"     "metadata"
46 [9] "source"                 "in_reply_to_status_id"
47 [11] "in_reply_to_status_id_str" "in_reply_to_user_id"
48 [13] "in_reply_to_user_id_str"   "in_reply_to_screen_name"
49 [15] "user"                  "geo"
50 [17] "coordinates"           "place"
51 [19] "contributors"          "is_quote_status"
52 [21] "retweet_count"          "favorite_count"
53 [23] "favorited"              "retweeted"
54 [25] "possibly_sensitive"      "lang"
55 > statuses[[1]][c("created_at", "text", "retweeted")]
56 $created_at
57 [1] "Thu Jan 09 12:06:27 +0000 2020"
58
59 $text
60 [1] "So yeah it's winter but there's no reason for being this cold \U00001f976
       (-13 degrees fahrenheit) https://t.co/3CkLjxo6TI"
61
62 $retweeted
63 [1] FALSE

```

Listing 7: Create Authentication Token and 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:

```

1 > #Endpoint URL:
2 > eURL<-"https://api.twitter.com/1.1/search/tweets.json?q=Winter&count=100"
3 >
4 > #Send the GET Request
5 > req <- GET(eURL,token)
6 >
7 > #Convert any http request errors into R errors
8 > stop_for_status(req)
9 >
10 > #Grab the Response Data. R will return it as a list rather than in JSON

```

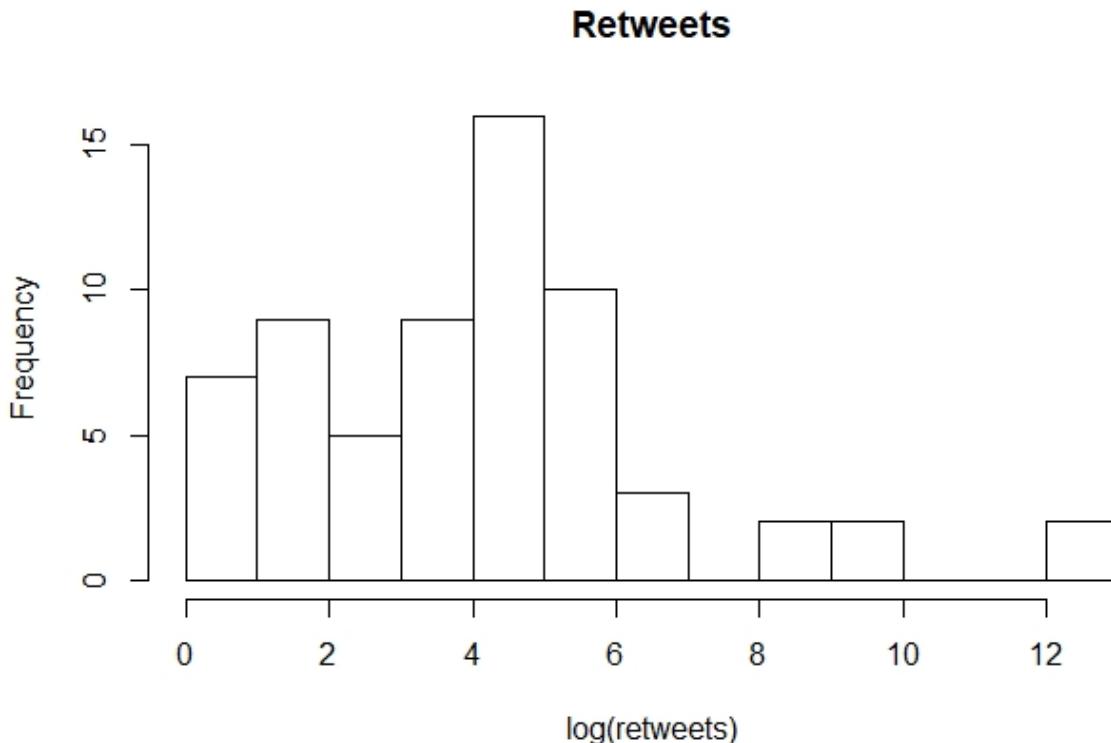
```

11 > response<-content(req)
12 >
13 > #See the structure of the response:
14 > names(response)
15 [1] "statuses"           "search_metadata"
16 >
17 > #We only care about the Twitter statuses:
18 > statuses<-response$statuses
19 >
20 > #Use lapply to iterate through each tweet in the list, and have it pull only
21 > #the retweet_count property from the tweet. Then, unlist it to convert it to
22 > #a vector of retweets.
23 > retweets<-unlist(lapply(statuses,function(x){x$retweet_count}))
24 >
25 > summary(retweets)
26   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
27     0.0     0.0     6.5    9040.2    110.8 433008.0
28
29 > 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:

```

1 > #Create and Authentication Header
2 > auth_header <- add_headers('Authorization'= paste('Bearer', apiKey))
3 >
4 > #Construct the URL. Recommended to use Insomnia first to ensure spaces are
   proper
5 > url<- "https://api.yelp.com/v3/businesses/search?term=Pizza&location=Wayne,%20NJ"
6 >
7 > #Send it off!
8 > rep<-GET(url,auth_header)
9 >
10 > #Grab Response Data:
11 > body_rep<-content(rep)
12 >
13 > #Inspect Structure:
14 > names(body_rep)
15 [1] "businesses" "total"      "region"
16 >
17 > #Inspect Business Structure:
18 > names(body_rep[["businesses"]][[1]])
19 [1] "id"           "alias"        "name"         "image_url"    "is_closed"
20 [6] "url"          "review_count" "categories"   "rating"      "coordinates"
21 [11] "transactions" "price"       "location"     "phone"       "display_
   phone"
22 [16] "distance"
23 >
24 > #Get all business names that returned
25 > b_names<-unlist(lapply(body_rep[["businesses"]],function(x){x$name}))
26 > b_names
27 [1] "Pizza One"
28 [2] "Pie Guys Pizza"
29 [3] "Neil's Pizzeria"
30 [4] "Coney Island Wood Fired Pizza"
31 [5] "Vinni's Pizzarama"
32 [6] "Buonocores brick oven pizza, panini, and mozzer"
33 [7] "Positano Restaurant & Pizzeria"
34 [8] "Amore's Pizza & Pasta"
35 [9] "Buongusto Pizza Restaurant & Catering"
36 [10] "Dominick's Pizzeria"
37 [11] "Se7te Wood Fire Pizza"
38 [12] "Uncle Louie's Pizza"
39 [13] "Gencarelli's"
40 [14] "Taste Of Tuscany"
41 [15] "Aquila Pizza Al Forno"
42 [16] "Time Out Cafe"
43 [17] "Brother Bruno's Pizza, Deli & Bagels"
44 [18] "Bradlees Pizzeria"
45 [19] "Anthony's Coal Fired Pizza"
46 [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:

```

1 > #Create and Authentication Header. Note that we also change the Content-Type
   parameter in the header:
2 > auth_header <- add_headers('Authorization'= paste('Bearer', apiKey),
3 +                               'Content-Type' = "application/graphql"
4 +
5 >
6 > #Construct the URL. This is just the endpoint for the graphql on the server
7 > url<- "https://api.yelp.com/v3/graphqli"
8 >
9 > #Construct the GraphQL Query in JSON:
10 > body<-|{
11 +   search(term: \"universities\",
12 +         location: \"wayne, nj\",
13 +         limit: 5) {
14 +   total
15 +   business {
16 +     name
17 +     url
18 +     location {
19 +       city
20 +       state
21 +     }
22 +   }
23 + }
24 + }"
25 >
26 >
27 > #Send it off!
28 > rep<-POST(url,auth_header,body=body)
29 >
30 > #Grab Response Data:
31 > body_rep<-content(rep)
32 >
33 > #Inspect Structure:
34 > names(body_rep)
35 [1] "data"
36 >
37 > #Inspect data Structure:
38 > names(body_rep[["data"]])
39 [1] "search"
40 >
41 > #Inspect search structure:
42 > names(body_rep[["data"]][["search"]])
43 [1] "total"      "business"
44 >
45 > #Get list of businesses:
46 > businesses<-body_rep[["data"]][["search"]][["business"]]
47 >
48 > #Inspect properties of single business:
49 > names(businesses[[1]])

```

```

50 [1] "name"      "url"       "location"
51 >
52 > #Get all business names and cities that returned
53 > b_names<-unlist(lapply(businesses,function(x){c(x$name)}))
54 > b_urls<-unlist(lapply(businesses,function(x){c(x$location$city)}))
55 >
56 > #Now construct a data frame with our data that we pulled from the API:
57 > businesses<-data.frame(business_name = b_names,city = b_urls)
58 >
59 > businesses
60   business_name      city
61 1 William Paterson University      Wayne
62 2 Montclair State University    Montclair
63 3          Caldwell College    Caldwell
64 4        C2 Education of Wayne      Wayne
65 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:

```

1 > library(rtweet)
2 > #Create the Token
3 > token<- create_token(
4 +   app="BANDMDW",
5 +   consumer_key="Ry1J8unRBomSm16PyDhcjv1c0",
6 +   consumer_secret = "Tn40ntfHhd53mzNRq2bPiwSCeRYhHQz7CneK904uBJsjwktN6k",
7 +   access_token = "1214308773323366400-aokPY8aPbVjxf29f07FktuceDOPHd3",
8 +   access_secret= "w7jyjBd5iD102paypCCivRGdkI4wD4J134F4Jzb0ZqpGe")
9 >
10 > #Search for tweets with the keyword "Winter".
11 > tweets<-search_tweets("Winter",token=token)
12 >
13 > #Convert to a data frame
14 > tweets<-data.frame(tweets)
15 >
16 > #Inspect the names in the data frame

```

```

17 > names(tweets)
18 [1] "user_id"                      "status_id"
19 [3] "created_at"                   "screen_name"
20 [5] "text"                         "source"
21 [7] "display_text_width"           "reply_to_status_id"
22 [9] "reply_to_user_id"             "reply_to_screen_name"
23 [11] "is_quote"                    "is_retweet"
24 [13] "favorite_count"              "retweet_count"
25 [15] "quote_count"                 "reply_count"
26 [17] "hashtags"                   "symbols"
27 [19] "urls_url"                   "urls_t.co"
28 [21] "urls_expanded_url"          "media_url"
29 [23] "media_t.co"                  "media_expanded_url"
30 [25] "media_type"                  "ext_media_url"
31 [27] "ext_media_t.co"              "ext_media_expanded_url"
32 [29] "ext_media_type"              "mentions_user_id"
33 [31] "mentions_screen_name"        "lang"
34 [33] "quoted_status_id"            "quoted_text"
35 [35] "quoted_created_at"           "quoted_source"
36 [37] "quoted_favorite_count"       "quoted_retweet_count"
37 [39] "quoted_user_id"              "quoted_screen_name"
38 [41] "quoted_name"                 "quoted_followers_count"
39 [43] "quoted_friends_count"         "quoted_statuses_count"
40 [45] "quoted_location"              "quoted_description"
41 [47] "quoted_verified"              "retweet_status_id"
42 [49] "retweet_text"                 "retweet_created_at"
43 [51] "retweet_source"                "retweet_favorite_count"
44 [53] "retweet_retweet_count"         "retweet_user_id"
45 [55] "retweet_screen_name"          "retweet_name"
46 [57] "retweet_followers_count"       "retweet_friends_count"
47 [59] "retweet_statuses_count"        "retweet_location"
48 [61] "retweet_description"           "retweet_verified"
49 [63] "place_url"                   "place_name"
50 [65] "place_full_name"              "place_type"
51 [67] "country"                     "country_code"
52 [69] "geo_coords"                  "coords_coords"
53 [71] "bbox_coords"                  "status_url"
54 [73] "name"                         "location"
55 [75] "description"                  "url"
56 [77] "protected"                    "followers_count"
57 [79] "friends_count"                 "listed_count"
58 [81] "statuses_count"                "favourites_count"
59 [83] "account_created_at"            "verified"
60 [85] "profile_url"                  "profile_expanded_url"
61 [87] "account_lang"                  "profile_banner_url"
62 [89] "profile_background_url"        "profile_image_url"
63 >
64 > #Only grab the screen name, tweet creation timestamp, and retweet count for each
   tweet returned
65 > tweets[,c("screen_name","created_at","retweet_count")]
66   screen_name      created_at retweet_count
67 1   eleanorquiroz 2020-01-09 13:40:32      6
68 2     Annexcali 2020-01-09 13:40:31      0
69 3     clapa238 2020-01-09 13:40:31      1
70 4    ShanghaiEye 2020-01-09 13:40:31      0
71 5    winter_fuego 2020-01-09 13:40:30    3164

```

72	6	allneworddotie	2020-01-09	13:40:30	0
73	7	ymzrspkc	2020-01-09	13:40:30	0
74	8	MarcJaylynn	2020-01-09	13:40:30	2
75	9	S3roTonym	2020-01-09	13:40:29	0
76	10	ChSuptRoads	2020-01-09	13:40:28	7
77	11	RHeightsFinest	2020-01-09	13:40:27	0
78	12	jbonnyman	2020-01-09	13:40:27	5
79	13	cutie_icee	2020-01-09	13:40:27	78
80	14	citybhlight67	2020-01-09	13:40:27	14
81	15	winter_flower3	2020-01-09	13:40:26	0
82	16	friendoftoads	2020-01-09	13:40:26	0
83	17	LOLO_0526	2020-01-09	13:40:26	38
84	18	SummerBreezeUS	2020-01-09	13:40:25	1
85	19	pan411170	2020-01-09	13:40:25	11
86	20	zwaantje1983	2020-01-09	13:40:24	0
87	21	winter_baby_	2020-01-09	13:40:24	24
88	22	to_themoon_0	2020-01-09	13:40:24	3905
89	23	BTSNJY1230	2020-01-09	13:40:23	13
90	24	PLaosupho	2020-01-09	13:40:23	159
91	25	Cxo_diana	2020-01-09	13:40:22	9060
92	26	robertmacIntos3	2020-01-09	13:40:22	20
93	27	JKHOPE_twt	2020-01-09	13:40:21	10
94	28	GlenviewILPD	2020-01-09	13:40:21	0
95	29	cH06SNWXbhU30pu	2020-01-09	13:40:20	217
96	30	uwhie_RCL	2020-01-09	13:40:20	433123
97	31	BABY_LEO_BILL	2020-01-09	13:40:20	180
98	32	lemonSVTcarat	2020-01-09	13:40:19	72
99	33	HOB1HOB1_S2	2020-01-09	13:40:19	0
100	34	DiscoverNEPA	2020-01-09	13:40:19	0
101	35	donuts6666	2020-01-09	13:40:18	67
102	36	Bom_Winter	2020-01-09	13:40:18	33
103	37	Aarti_Panday	2020-01-09	13:40:18	69
104	38	Vibezzz_	2020-01-09	13:40:17	3068
105	39	pcsendashonga	2020-01-09	13:40:17	5
106	40	Kat_Missouri	2020-01-09	13:40:17	4
107	41	Faust89823700	2020-01-09	13:40:15	3
108	42	IrisOfTheBronx	2020-01-09	13:40:15	2
109	43	markwinder8	2020-01-09	13:40:15	1
110	44	GrandsonofRickk	2020-01-09	13:40:14	201
111	45	Ariluminosa	2020-01-09	13:40:14	0
112	46	Empress_Marsha	2020-01-09	13:40:13	0
113	47	mistrasparta	2020-01-09	13:40:13	0
114	48	AutosourcePtbo	2020-01-09	13:40:13	0
115	49	eeaaaakkkk	2020-01-09	13:40:13	0
116	50	ggukmei	2020-01-09	13:40:12	5532
117	51	hosoku_is_myliif	2020-01-09	13:40:11	31
118	52	maxtheultracat	2020-01-09	13:40:10	0
119	53	puadekplaylist	2020-01-09	13:40:09	268
120	54	OyekanmiCollar	2020-01-09	13:40:08	3
121	55	XRP_589_Theunis	2020-01-09	13:40:08	1
122	56	Leonor93726311	2020-01-09	13:40:08	1522
123	57	1stEdCU	2020-01-09	13:40:08	0
124	58	morrelife	2020-01-09	13:40:08	581
125	59	winter_pokeca	2020-01-09	13:40:06	0
126	60	hrdrock2	2020-01-09	13:40:05	67
127	61	8AvLFe5JwBhiC0m	2020-01-09	13:40:05	3905

128	62	Imskey	2020-01-09	13:40:05	0
129	63	AndrewRussell15	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	hassouuunii	2020-01-09	13:39:58	3
163	97	mdbaumbach	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".