Getting Data using APIs

Melissa Bischoff EDAV Community Contribution

Why use APIs?

API integrations are an easy and useful way to pull data from the internet before resorting to web scraping. A lot of companies and services have APIs available, some are free for public use and some are for paying customers.

What is an API?

API stands for application programming interface. It is a way to retrieve, edit, or delete data from an endpoint. API's are extremely common in the tech industry, they are used within companies and between companies. Publicly available, free endpoints to retrieve data from are what I will focus on in this tutorial.

What is a RESTful API?

REST stands for representational state transfer and it is an API structure. A RESTful API (AKA a REST API) is an API that conforms to the constraints of REST architectural style. The constraints are well-documented on the internet but are not necessary to dive deep into here.

HTTP Request Methods

HTTP requests are what contact the API server to make a secure connection. There are four types of requests; they are explained in the table below.

Request	**Usage**
GET	Retrieve resource information only. (will use to get datasets)
POST	Create new resources. (will use to create an access token for authorization)
PUT	Update existing resources.
DELETE	Delete existing resources.
PATCH	Partially update existing resources.

In this tutorial, since we are focused on retrieving data from an API, we will focus on the GET method. When we do an example that requires authorization we will use the POST method as well.

Authorization

An important piece of information when using API's is the authorization flow. Basically, this is how you are allowed to access an API to retrieve data. Authorization flows are specific to each API. Authorization information is typically found in the API's documentation that you're using. Sometimes there is no authorization required. I will do two examples, one where it is required and one where it isn't.

Github Job Posting API Example in R

Github has a free API for downloading all of the job postings on their website. Their API does not require authorization. We will access the API to download information on remote data-related jobs.

necessary packages:

```
library(httr)
library(jsonlite)
library(seqinr)
library(base64enc)
library(dplyr)
```

Make a GET request to the API

```
response = httr::GET("https://jobs.github.com/positions.json?description=api")
```

Here we used the `httr` package's function `GET` to retrieve a response from the github API. Since there is no authentication necessary, we can simply request the URL. You can find the url to an API by just googling it.

A request object
This returns a response object with information about what we requested. We saved the response object as the variable `response` so we can inspect it further.

```
response
```

```
## Response [https://jobs.github.com/positions.json?description=api]
##   Date: 2021-03-09 18:12
##   Status: 200
##   Content-Type: application/json; charset=utf-8
##   Size: 311 kB
```

There are a few things stored in the response object. These are standard when making any request.

`response$times` returns the time it took to transfer the data, etc.:

```
##      redirect    namelookup       connect   pretransfer starttransfer
##      0.000000      0.001367      0.065928      0.194273      0.975535
##         total
##      1.223549
```

`response$url` returns the URL that we requested originally:

```
response$url
```

```
## [1] "https://jobs.github.com/positions.json?description=api"
```

`response$status_code` returns the status code of the request. This code is also displayed when we return the `response` object. A status code will tell us the result of our request based on what method we used. Codes in the 400s indicate that an error occurred. Most codes in the 200s mean the request succeeded. It is very easy to google your code to find out what it means. Here is a site I like to use to lookup status codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/ We got a `status_code` 200 which means OK. This means that our GET request has succeed and the resource has been obtained:

```
response$status_code
```

```
## [1] 200
```

`response$headers` gives more detailed context about the response. These are different for different URLs, so these are specific to the Github API. The headers are not necessary for this example so I will not include the output.

`response$cookies` returns an HTTP cookie. This is a piece of data that a server sends to the user's web browser. We did not send any cookies so this returns nothing.

`response$date` returns the date of the request:

```
response$date
```

```
## [1] "2021-03-09 18:12:45 GMT"
```

`response$request` returns more information about our request:

```
response$request
```

```
## <request>
## GET https://jobs.github.com/positions.json?description=api
## Output: write_memory
## Options:
## * useragent: libcurl/7.54.0 r-curl/4.3 httr/1.4.2
## * httpget: TRUE
## Headers:
## * Accept: application/json, text/xml, application/xml, */*
```

All of the above options that we can collect from the response object are pretty standard. `Content` is another standard output that returns more context on the request. If the GET request was successful (`status_code = 200`) then the `content` will return the resource requested. If there is a `status_code` representing an error, `content` will return more information on the error. Thus, it is useful for de-bugging bad requests.

Cleaning & using the API data

`response$content` gives us the content of the request, in this case it is the data that we requested. It is in an ecrypted JSON format. We know this because of the output of the `response` object returns

Content-Type: application/json; charset=utf-8. This means it is in json format but ecrypted utf-8. To decrypt the data, we use the function rawToChar. To convert the JSON to a data frame we use the fromJSON function from the jsonlite package. It is very typical for data from an API to be returned (and requests sent) in JSON form, thus the jsonlite package is very handy when making API requests. R does not have a built-in data type for JSON objects whereas Python does (more on this later).

```
raw_data = rawToChar(response$content)
data = data.frame(fromJSON(raw_data))
```

Now we have our data in a dataframe and we can explore what we have in it:

```
colnames(data)
```

```
##  [1] "id"            "type"          "url"           "created_at"    "company"
##  [6] "company_url"   "location"      "title"         "description"   "how_to_apply"
## [11] "company_logo"
```

Now I'm able to easily access and use the data from the API. For example, I can find all the companies that are hiring for remote jobs and what the positions are.

```
remote_jobs = filter(data, !grepl("Remote", data$location))
remote_jobs %>% select(company, title)
```

```
##                                                    company
## 1                                                  presize
## 2                                         Huey Magoo's LLC
## 3                                                  SovTech
## 4                                                Microsoft
## 5  MANDARIN MEDIEN Gesellschaft für digitale Lösungen mbH
## 6                                                Saildrone
## 7                                      Football Addicts AB
## 8                                              Boxine GmbH
## 9                                               Flowmailer
## 10                                              azeti GmbH
## 11                                           Comma Soft AG
## 12                                        Commonwealth Bank
## 13                                                Alliander
## 14                                        Commonwealth Bank
## 15                                               Schüttflix
## 16                                         TeleClinic GmbH
## 17                                         TeleClinic GmbH
## 18                               LORENZ Life Sciences Group
## 19                                                  Snappet
## 20                       Shell Business operations, Chennai
## 21                                              Sono Motors
## 22                                            Agiloft, Inc
## 23                                          Boston Red Sox 
## 24                                    ALD AutoLeasing D GmbH
## 25                             HUK-COBURG Autoservice GmbH
## 26                             HUK-COBURG Autoservice GmbH
## 27                                        Tokyo Digital Ltd
## 28                        Cornell University - Breeding Insight
## 29                                 The Nature Conservancy
## 30                                     McKinsey & Company
## 31                                     McKinsey & Company
## 32                                     McKinsey & Company
## 33           Foundation for Interwallet Operability (FIO)
## 34                                           madewithlove
## 35                                            DSPolitical
## 36                                                Percona
##                                                                 title
## 1                            Software Engineer, Front End (m/f/d)
## 2                                                   IT Specialist
## 3                                    Javascript Software Engineer
## 4                       Microsoft Software Engineer. Dublin, Ireland.
## 5                                              Frontend Developer
## 6              Senior Software Engineer - Vehicle Command & Control
## 7                                Senior Backend Developer (Remote)
## 8                 Softwareentwickler Backend (PHP/Symfony) (m/w/d)
## 9                                               Software Engineer
## 10                             Lead Developer - azeti MES Platform
## 11                          (Senior) Backend Developer .Net (m/w/d)
## 12            Back End Senior Software Engineer (.Net, .Net Core)
## 13                                Lead Data Engineer bij Alliander
## 14                     Senior Data Engineer - Scala, Spark, Big Data
## 15          Quality Assurance Engineer (m/w/d), Köln oder Gütersloh
## 16         Python / Django Developer (f/m/d) *100% remote possible*
## 17               Fullstack Developer (f/m/d) *100% remote possible*
## 18                                Automated Testing Expert (m/w/d)
## 19                                            Full-stack Developer
## 20                                  Data Science - Team Lead (CRM)
## 21 Mobile App Entwickler (w/m/x) für SonoDigital Mobility Services
## 22                     Integrations Developer - Salesforce (Remote)
## 23                                  Developer, Baseball Systems 
## 24     DevOps Engineer (f/d/m) / Site Reliability Engineer (f/d/m)
## 25                                      Backend-Entwickler (w/m/d)
## 26                                     Frontend-Entwickler (w/m/d)
## 27                                       Technical Project Manager
## 28                                          Applications Developer
## 29                                                   Web Developer
## 30                             Capabilities and Insights Analyst
## 31                                              Software Engineer
## 32                                                  Data Engineer
## 33                                  QA and Test Automation Engineer
## 34                                     Full-stack engineer (f/m/x)
## 35                                            Front End Engineer I
## 36                               Golang Software Engineer (remote)
```

I can also get the links to the applications quickly.

```
remote_data_jobs = filter(data, !grepl("Data", data$title))
job_links_to_apply_to = remote_data_jobs$how_to_apply
```

API Integration in Python

I personally like using Python over R for API integrations. The `requests` module is very well documented and the returned object has more options for viewing the results than in R. Additionally, Python handles JSON object much better than R since it has `dict` data types and R does not. JSON objects are very commonly used when sending and receiving data with APIs so having built-in `dict` datatypes, which are the same format as JSON objects, is extremely useful.

The Github Job Posting Example

Here is the code to make the same request in Python using the `requests` module.

import packages:

```
In [1]:  import requests
         import base64
         import pandas as pd
```

```
In [2]:  response = requests.get("https://jobs.github.com/positions.json?description=api"
```

In Python, the response object returns the response and status code. We have a code 200, so the retrieval was a success.

```
In [3]:  response

Out[3]:  <Response [200]>
```

The `requests` package has more options than the `httr` package in R. Since we have a code 200, we are ready to look at the content of the returned data. Requests returns the data in JSON (`response.json()`), text (`response.text`), or raw format (`response.content`). Having multiple options of datatype output is useful depending on how you want to use the data. Python's `dict` objects read JSON well so I will save the response content as a json object.

```
In [4]:  json_data = response.json()
```

Below is an example of how a `dict` and JSON object are structured. It is in the form of key and value pairs. For exampe, the key is `id` and the value is `b1008413-2d80-49cf-be28-f784d3978788` . This is the job ID for first job posting I printed below. In the second line, the key is `type` and the value is `Full Time` . This means that the job type for this first job posting is full time. Each value in this dataset has these keys and values.

```
In [27]:  json_data[3]

Out[27]:  {'id': 'b1008413-2d80-49cf-be28-f784d3978788',
           'type': 'Full Time',
           'url': 'https://jobs.github.com/positions/b1008413-2d80-49cf-be28-f784d397878
          8',
           'created_at': 'Mon Mar 08 15:13:54 UTC 2021',
           'company': 'SovTech',
           'company_url': 'https://ltpx.nl/lubbp4y',
           'location': 'Johannesburg',
 'title': 'Javascript Software Engineer',
 'description': '<p><strong>Javascript Software Engineer</strong></p>\n<p>At Sov
Tech we design, build, deploy and maintain innovative custom software that gives
our clients the opportunity to start, run and grow world class businesses We are
currently on the lookout for Software Engineers to join our team on a variety of
projects that are kicking off over the next few months. Our teams are rapidly gr
owing and we are looking for Developers that are passionate about building softw
are for our World class clients. Firstly...We LOVE Javascript.</p>\n<p><strong>J
ob requirements</strong>\nWe are looking for Javascript Developers at ALL levels
in all regions. Our head quarters are in Johannesburg, South Africa.\xa0</p>\n<p
>We have\xa0Guilds situated\xa0in Cape Town, Johannesburg &amp; London.</p>\n<p>
If you have experience working with ReactJS, React Native, Nodejs, Typescript &a
mp; AWS specifically, you might be what we have been searching for!\xa0 Our Tech
nology Stack,\xa0We build almost everything on the Serverless framework with AWS
behind the scenes. We love React on the front, web and mobile.\xa0</p>\n<p>SovTe
ch Tech Stack:\xa0\n<strong>Backend:</strong></p>\n<ul>\n<li>GraphQL</li>\n<li>R
EST</li>\n<li>NodeJS</li>\n</ul>\n<p><strong>Frontend:</strong></p>\n<ul>\n<li>R
eactJS</li>\n<li>React Native</li>\n<li>Typescript</li>\n<li>CSS-in-JS (Styled C
omponents)</li>\n<li>Tests first</li>\n</ul>\n<p><strong>Cloud Services:</strong
>\nNumerous AWS services:</p>\n<ul>\n<li>AppSync</li>\n<li>Cloudfront</li>\n<li>
CloudWatch</li>\n<li>Cognito</li>\n<li>Lambda</li>\n<li>DynamoDB</li>\n<li>S3</l
i>\n<li>SES</li>\n<li>SNS</li>\n<li>SQS</li>\n</ul>\n<p>And many more!</p>\n<p><
strong>Other notable tech things</strong></p>\n<ul>\n<li>Bitbucket + JIRA + Slac
k</li>\n<li>Git-Flow</li>\n<li>Monorepo approach</li>\n<li>AWS CDK</li>\n<li>Ser
verless framework</li>\n<li>Docker + Docker-Compose</li>\n<li>Create React App</
li>\n<li>Industry best practices + standards</li>\n</ul>\n<p><strong>What cool t
hings do we offer at SovTech?</strong></p>\n<ul>\n<li>Annual full-company X-mas
retreat</li>\n<li>Dev chats every Friday with a beer or dial in from where ever
you are based in the world</li>\n<li>Do you enjoy Soccer? Join our SovTech Stars
Soccerclub</li>\n<li>Do you enjoy running? Join our SovTech running Club</li>\n<
li>Slack channels like #Memes, #Geekingitup, #10o’clockrock, #AmazingDesigns #Hu
mansofSovTech that connects you to our network of awesome people</li>\n<li>Hatch
(Annual company-wide hackathon)</li>\n<li>Annual FoosFestival with our very own
Minister of Foosball</li>\n<li>Our own currency(Stacos) - 50 Stacos is given to
each employee every month by the company to spend on rewarding and recognising y
our colleagues, Stacos are redeemable for a variety of online shopping vouchers
</li>\n</ul>\n<p><strong>Remote vs On-site</strong>\nWe believe in teams. Our te
ams are often distributed but everybody has a home base. We call it a Guild. At
present, you must be in Johannesburg, London, or Cape Town. How else do we have
a beer on a Friday together? But yes you can also work from home when you want.
Is this you? Join us! Apply via the\xa0<strong>application button</strong>.</p>
\n<p><em>Agency calls are not appreciated.</em></p>\n',
 'how_to_apply': '<p><a href="https://ltpx.nl/zhMQLcU">Click to apply</a></p>
\n',
 'company_logo': 'https://jobs.github.com/rails/active_storage/blobs/eyJfcmFpbHM
iOnsibWVzc2FnZSI6IkJBaHBBcmViIiwiZXhwIjpudWxsLCJwdXIiOiJibG9iX2lkIn19--a861f5844
a9014b5488296389351c9eebdae258f/sovtech_logo-2.png'}
```

The reason these `dict` structured data are so convenient in Python is because `Pandas` (and other modules) can easily convert them into dataframes. Each key becomes a column name and the values for the keys form a row for 1 entry. Below I printed the same job posting as above. We can see how the keys and values fall into columns and rows.

In [29]:
```python
jobs_df = pd.DataFrame(json_data)
jobs_df[3:4]
```

Out[29]:

	id	type	url	created_at	company	co
3	b1008413-2d80-49cf-be28-f784d3978788	Full Time	https://jobs.github.com/positions/b1008413-2d8...	Mon Mar 08 15:13:54 UTC 2021	SovTech	https://ltpx

A more advanced example in Python using the Spotify API and access tokens

For this example, we will use the Spotify Web API. I picked this API because it is easy to access (anyone with a Spotify account can get an access token) and the API resources and authorization flow are well documented. It also has a TON of interesting data - **potential final project data source!**

Information on the data available and how to retrieve it via the API is found here: https://developer.spotify.com/documentation/web-api/reference/#reference-index

This example differs from the last as we now need to send additional data with our request, our authorization credentials. Authorization information is found here: https://developer.spotify.com/documentation/general/guides/authorization-guide/

We see in the documentation that we need to first create and store an access token from the API using the `POST` method. Once we create an access token we will use it in our `GET` request to get the data. We have to send our client_id and client_secret with the request in order to create an access token. The docs say that it has to be in the format `<base64 encoded client_id:client_secret>`.

The client id and client secret are given to you when creating an application account at https://developer.spotify.com/dashboard/login

Note that I redacted my keys from the code so that they cannot be re-used

In [8]:
```
secret_bytes = bytes(('{}:{}'.format(REDACTED_CLIENT_ID, REDACTED_CLIENT_SECRET)
secret_enc = base64.b64encode(secret_bytes).decode('utf-8')
```

Now that we have encrypted our id and secret, we can build the `POST` request function `requests.post()`. The requests function takes the arguments `url` of the API endpoint, `headers` which contain more information about the resource to be fetched or, in this case, about the client requesting the resource, and `data` which also contains more information to send along with the request. In the `requests` module, the `headers` and `data` are sent in `dict` format (another reason why the built-in `dict` datatype makes requests so easy in Python).

`requests.post()` returns a response object that we will store into the variable `response_post`.

In [9]:
```
data = {'grant_type': 'client_credentials'}
headers = {'Authorization': 'Basic {}'.format(secret_enc)}
url = 'https://accounts.spotify.com/api/token'
response_post = requests.post(url, headers=headers, data=data)
```

Building the request can be confusing - here is a cool cheat

Sometimes it is difficult to figure out how to build the `request` function, what goes into `headers` and `data` arguments, etc.

Often times, well-documented APIs will have examples of the `cURL` requested function to use to make the request you want. `cURL` is the base language that makes the requests; packages and modules like `httr` in `R` and `requests` in `Python` basically make `cURL` accessible in these languages. Other packages for other languages are available as well.

find the `cURL` command:

The Spotify documentation lists a `cURL` sample command, it is: `curl -X "POST" -H "Authorization: Basic ZjM4ZjAw...WY0MzE=" -d grant_type=client_credentials https://accounts.spotify.com/api/token` It'll start with `curl` so you can do a CMD+F

insert the command into this `cURL` converter:

https://onlinedevtools.in/curl

Copy, paste, and select the language you want the output command to be. The output returns what to put into `data` and `header` arguments and how to run the `requests.post()` command - is the same as our code above. *but note that you have to enter your encrypted client ID and secret - do not copy the example ones as they are not valid.*

We should check that our request was successful and we receieved a `status_code = 200` , then we can save our access token in the variable `tk` . The token is stored in the content of the response.

In [10]:
```
if response_post.status_code == 200:
    tk = response_post.json()['access_token']
else:
    print('Something went wrong, status_code = {}'.format(response_post.status_c
```

Now we can build our `GET` request to retrieve the Spotify data. There is a lot of data that we can get from the Spotify API (can read more about the data available here: https://developer.spotify.com/documentation/web-api/reference/#reference-index).

I want to get information on audio features of some songs. More information on getting audio features for tracks here: https://developer.spotify.com/documentation/web-api/reference/#endpoint-get-several-audio-features

From the documentation, we see that we need to send our access token and Spotify track IDs of the songs that we want features on with our request. We have our access token from our `POST` request above. We read in the docs that each Spotify song has an ID associated with it, these are easily found on the Spotify app or web player (can google how to find the Spotify ID for a song).

The Spotify documentation provides an example of a `cURL` `GET` request for getting audio features. We can use it to create our `requests.get()` function by inserting it into the `cURL` converter explained above (https://onlinedevtools.in/curl).

sample `cURL` request from Spotify documentation on getting audio features:

`curl -X "GET" "https://api.spotify.com/v1/audio-features?ids=4JpKVNYnVcJ8tuMKjAj50A%2C24JygzOLM0EmRQeGtFcIcG" -H "Accept:`

```
application/json" -H "Content-Type: application/json" -H "Authorization: Bearer ACCESS_TOKEN"
```

To create the `requests.get()` fuction I will use some Spotify IDs from songs that I previously got via the Spotify API. We will also need our access token we got from our `POST` request and stored into `tk` above. I will store the `GET` response in the variable `response_get` .

In [11]:
```
# data I collected from the Spotify API previously and stored into a dict
# 'ID' = Spotify song ID
# 'name' = song name
# 'artist' = song artist
songs = [
    {'id':'3HAgxyWGeJtIVabS2mTREt',
     'name':'Vagabond',
     'artist':'Caamp'
    },
    {'id':'75nZ4W6quZhI55LKiqCXWh',
     'name':'By and By',
     'artist':'Caamp'
    },
    {'id':'6255IIBwKySv6RYrOeHfQh',
     'name':'All the Debts I Owe',
     'artist':'Caamp'
    },
    {'id':'4KhBvLbRr58rHPF24bdL9Q',
     'name':'Officer of Love',
     'artist':'Caamp'
    },
    {'id':'0ESvfrHfNuAtkZp8SMJBOY',
     'name':'Strawberries',
     'artist':'Caamp'
    },
    {'id':'2pfAvgMoHLfialvMYn337d',
     'name':'No Sleep',
     'artist':'Caamp'
    },
    {'id':'2m9ryxnEcVoQNr22KRxe09',
     'name':'Channel 43',
     'artist':'Deadmau5'
    },
    {'id':'4ua0IepBEISCWwF8dTJvcU',
     'name':'Ghosts n Stuff',
     'artist':'Deadmau5'
    },
    {'id':'2nLgWMdYPO35GGpwX2xo23',
     'name':'Monophobia',
     'artist':'Deadmau5'
    },
    {'id':'5cr1Daz7Kv03CNb51I18sy',
     'name':'Arguru 2k19"',
     'artist':'Deadmau5'
    },
    {'id':'7oPqRSubaWcDb5F68aw3P6',
     'name':'The Veldt',
     'artist':'Deadmau5'
    },
    {'id':'5emTyRnoMTLc9G44RR6XIE',
     'name':'Bridged By A Lightwave',
     'artist':'Deadmau5'
    }
]
```

In [12]:
```
# get only ids from data above
ids = []
for i in range(0,len(songs)):
    ids.append(songs[i]['id'])
```

In [13]:
```
# format the IDs for the request function ('ID,ID,ID')
ids_fmt = ','.join([str(x) for x in ids])
```

In [14]:
```
# make request code
headers = {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Authorization': 'Bearer {}'.format(tk),
}

params = (
    ('ids', ids_fmt),
)

response_get = requests.get('https://api.spotify.com/v1/audio-features', headers
```

Let's make sure we got a successful `status_code` and then store the JSON data into `audio_features`.

In [15]:
```
if response_get.status_code == 200:
    audio_features = response_get.json()['audio_features']
else:
    print('Something went wrong, status_code = {}'.format(response_get.status_co
```

We can run `response_get.url` to see the url we formed in our request. We see that it is the same format as the `cURL` in the example request where the IDs are separated by `%` signs.

In [16]:
```
response_get.url
```

Out[16]:
```
'https://api.spotify.com/v1/audio-features?ids=3HAgxyWGeJtIVabS2mTREt%2C75nZ4W6q
uZhI55LKiqCXWh%2C6255IIBwKySv6RYrOeHfQh%2C4KhBvLbRr58rHPF24bdL9Q%2C0ESvfrHfNuAtk
Zp8SMJBOY%2C2pfAvgMoHLfialvMYn337d%2C2m9ryxnEcVoQNr22KRxe09%2C4ua0IepBEISCWwF8dT
JvcU%2C2nLgWMdYPO35GGpwX2xo23%2C5cr1Daz7Kv03CNb51I18sy%2C7oPqRSubaWcDb5F68aw3P6%
2C5emTyRnoMTLc9G44RR6XIE'
```

In [17]:
```
audio_features_df = []
audio_features_rows = [i for i in audio_features if i]
for row in audio_features_rows:
    audio_features_df.append(row)

audio_features_df = pd.DataFrame(audio_features_df)
```

Here's what our data looks like:

In [18]:
```
audio_features_df[:3]
```

Out[18]:

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	live
0	0.571	0.399	4	-11.919	1	0.0364	0.766	0.014600	0.(

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	live
1	0.584	0.474	9	-7.981	1	0.0258	0.796	0.000003	0.
2	0.483	0.486	0	-11.062	1	0.0434	0.753	0.004790	0.

The columns are:

```
In [19]: list(audio_features_df.columns)
```

```
Out[19]: ['danceability',
          'energy',
          'key',
          'loudness',
          'mode',
          'speechiness',
          'acousticness',
          'instrumentalness',
          'liveness',
          'valence',
          'tempo',
          'type',
          'id',
          'uri',
          'track_href',
          'analysis_url',
          'duration_ms',
          'time_signature']
```

Details on what these numbers mean and how they are calculated are all found on Spotify's API documentation here: https://developer.spotify.com/documentation/web-api/reference/#endpoint-get-several-audio-features

Now we can plot some cool things with the data we recieved from the Spotify API. Let's first join in the initial data so we can have the artist and song name in the dataframe.

```
In [20]: songs_df = pd.DataFrame(songs)
```

```
In [21]: song_audio_features = pd.merge(
             audio_features_df,
             songs_df,
             on = ['id']
         )
```

```
In [23]: import seaborn as sns
         import matplotlib.pylab as plt
```

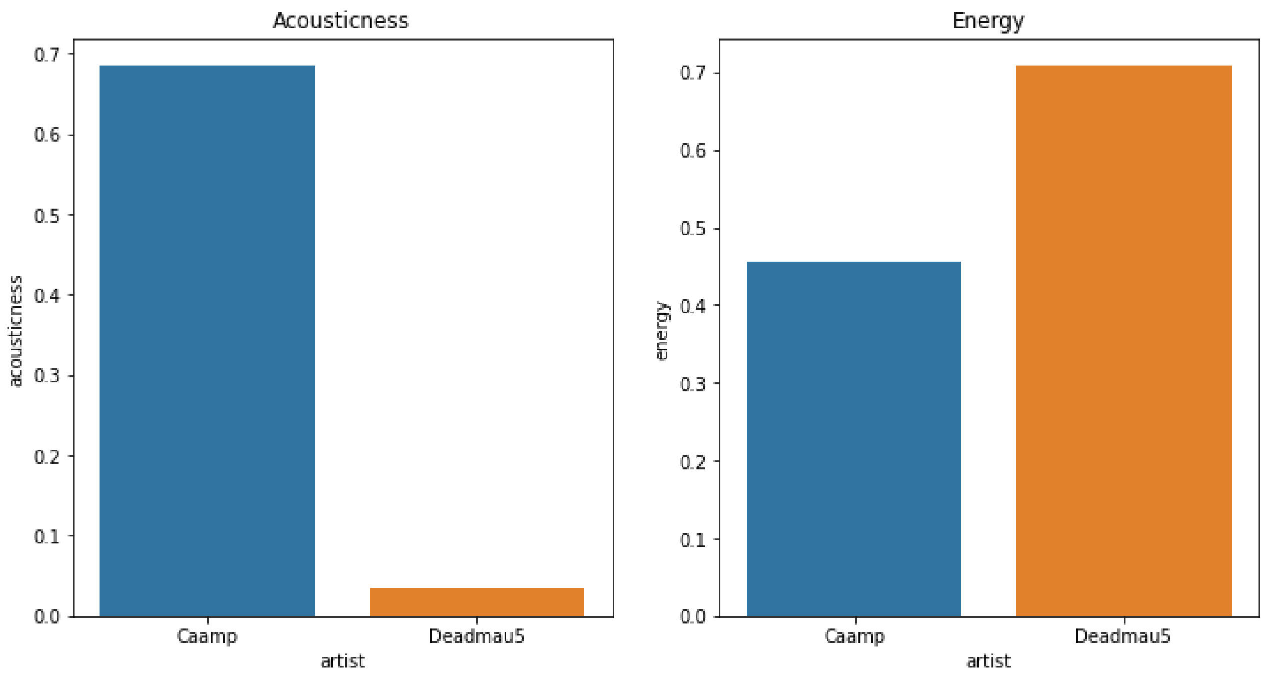
Let's compare the average audio features of each artist across their songs. For some context, Caamp is an acoustic chill band and Deadmau5 is an electronic band.

```
In [24]: grouped_artist = song_audio_features.groupby(by=['artist']).mean()
```

```
In [25]: fig,ax = plt.subplots(1,2, figsize = (12,6));

         sns.barplot(
             data = song_audio_features,
             y = 'acousticness',
             x = 'artist',
             ax = ax[0],
             ci = None
         );
         ax[0].set_title('Acousticness');

         sns.barplot(
             data = song_audio_features,
             y = 'energy',
             x = 'artist',
             ax = ax[1],
             ci = None
         );
         ax[1].set_title('Energy');
```



Caamp's music is much more acoustic, as expected. Whereas Deadmau5's music has much more energy.

A couple of notes:

- Read the API documentation thoroughly!
    - It will contain information on how to use the API. Sometimes it'll have very clear steps that you can follow.
- BUT not all API documentation are created equally!
    - Some APIs will have little or confusing documentation. In this case you can search the internet for additional information.
- Use the cURL converter - https://onlinedevtools.in/curl
    - Often times, API documentation will have samples of cURL commands to use with the request you want to make. You can convert the cURL into Syntax for R, Python, and other languages using this tool.

Hopefully these examples are a good starting point to using APIs to get data. Personally, I prefer using the requests module in Python to interact with APIs over the httr package in R. Information is available on both of these packages on the Python and R documentation. There are also many examples on Stack Overflow and other places on the internet of using these packages to make API requests.