**4/28->5/18/2021**

5/18/2022

6/7/22

* 4/7/21 🡪 4/13/21

1. Most important

* Coding.
* Python, Java, JavaScripts

1. API test

* Test strategy
* API test fundamentals

1. UI test

* Hands on selenium test
* Debug with browser tools.

1. Database tests

* Run complex queries

1. Linux environment
2. Automation

* Auto Framework
* Tools
* Generate data set
* Save logs/results
* Process/analyze data/results

1. CI/CD

API test

1. Retrieve Data

we’ll learn how to retrieve data for data science projects. There are millions of APIs online which **provide access to data**. Websites like [Reddit](https://www.reddit.com/dev/api/), [Twitter](https://developer.twitter.com/en/docs.html), and [Facebook](https://developers.facebook.com/) all offer certain data through their APIs.

But **why use an API** instead of a static CSV dataset you can download from the web? APIs are useful in the following cases:

* **The data is changing quickly**. An example of this is stock price data. It doesn’t really make sense to regenerate a dataset and download it every minute — this will take a lot of bandwidth, and be pretty slow.
* **You want a small piece of a much larger set of data**. Reddit comments are one example. What if you want to just pull your own comments on Reddit? It doesn’t make much sense to download the entire Reddit database, then filter just your own comments.
* **There is repeated computation involved**. Spotify has an API that can tell you the genre of a piece of music. You could theoretically create your own classifier, and use it to compute music categories, but you’ll never have as much data as Spotify does.

In cases like the ones above, an API is the right solution. In this blog post, we’ll be querying a simple API to retrieve data about the [International Space Station](https://en.wikipedia.org/wiki/International_Space_Station) (ISS).

* **Figure out what kind of data my API can offer**
* **The data could be public data and user’s private data**
* **How users can update their own data**

response = requests.get("http://api.open-notify.org/this-api-doesnt-exist")

The get() function returns a [**response object**](https://2.python-requests.org/en/master/user/advanced/#request-and-response-objects)**.** We can use the [response.status\_code](https://2.python-requests.org/en/master/user/quickstart/#response-status-codes) attribute to receive the status code for our request:

print(response.status\_code)

404

Often there will be multiple APIs available on a particular server. Each of these APIs are commonly called **endpoints**. The first endpoint we’ll use is [http://api.open-notify.org/astros.json](http://open-notify.org/Open-Notify-API/People-In-Space/), which returns data about astronauts currently in space.

JSON is the primary format in which data is passed back and forth to APIs, and most API servers will send their responses in JSON format.

import json

def jprint(obj):

# create a formatted string of the Python JSON object

text = json.dumps(obj, sort\_keys=True, indent=4)

print(text)

jprint(response.json())

{

"message": "success",

"number": 6,

"people": [

{

"craft": "ISS",

"name": "Alexey Ovchinin"

},

{

"craft": "ISS",

"name": "Nick Hague"

},

{

"craft": "ISS",

"name": "Christina Koch"

},

{

"craft": "ISS",

"name": "Alexander Skvortsov"

},

{

"craft": "ISS",

"name": "Luca Parmitano"

},

{

"craft": "ISS",

"name": "Andrew Morgan"

}

]

}

**API doc**

[**Open Notify (open-notify.org)**](http://open-notify.org/)

**How Many People Are In Space Right Now**

[**http://api.open-notify.org/astros.json**](http://api.open-notify.org/astros.json)

How many humans are in space *right now*?

**Overview**

This API returns the current number of people in space. When known it also returns the names and spacecraft those people are on. This API takes no inputs.

**Output**

JSON

[http://**api**.open-notify.org/astros.json](http://api.open-notify.org/astros.json)

{

"message": "success",

"number": NUMBER\_OF\_PEOPLE\_IN\_SPACE,

"people": [

{"name": NAME, "craft": SPACECRAFT\_NAME},

...

]

}

**Using an API with Query Parameters**

* **API end point use query to generate specific set of data.**
* **The query parameters can be used to filter, sort.**

It’s very common, however, to have an API endpoint that requires us to **specify parameters**. An example of this the [http://api.open-notify.org/iss-pass.json endpoint](http://open-notify.org/Open-Notify-API/ISS-Pass-Times/). This endpoint tells us the **next times** that the international space station will pass over a **given location** on the earth.

If we look at the documentation, it specifies required lat (latitude) and long (longitude) parameters.

We can do this by adding an optional keyword argument, params, to our request. We can make a dictionary with these parameters, and then pass them into the requests.get function. Here’s what our dictionary would look like, using coordinates for New York City:

parameters = {

"lat": 40.71,

"lon": -74

}

We can also **do the same thing directly by adding the parameters directly to the URL**. like this: http://api.open-notify.org/iss-pass.json?lat=40.71&lon;=-74.

It’s almost always preferable to setup the parameters as a dictionary, because requests takes care of some things that come up, like properly formatting the query parameters, and we don’t need to worry about inserting the values into the URL string.

Let’s make a request using these coordinates and see what response we get.

response = requests.get("http://api.open-notify.org/iss-pass.json", params=parameters)

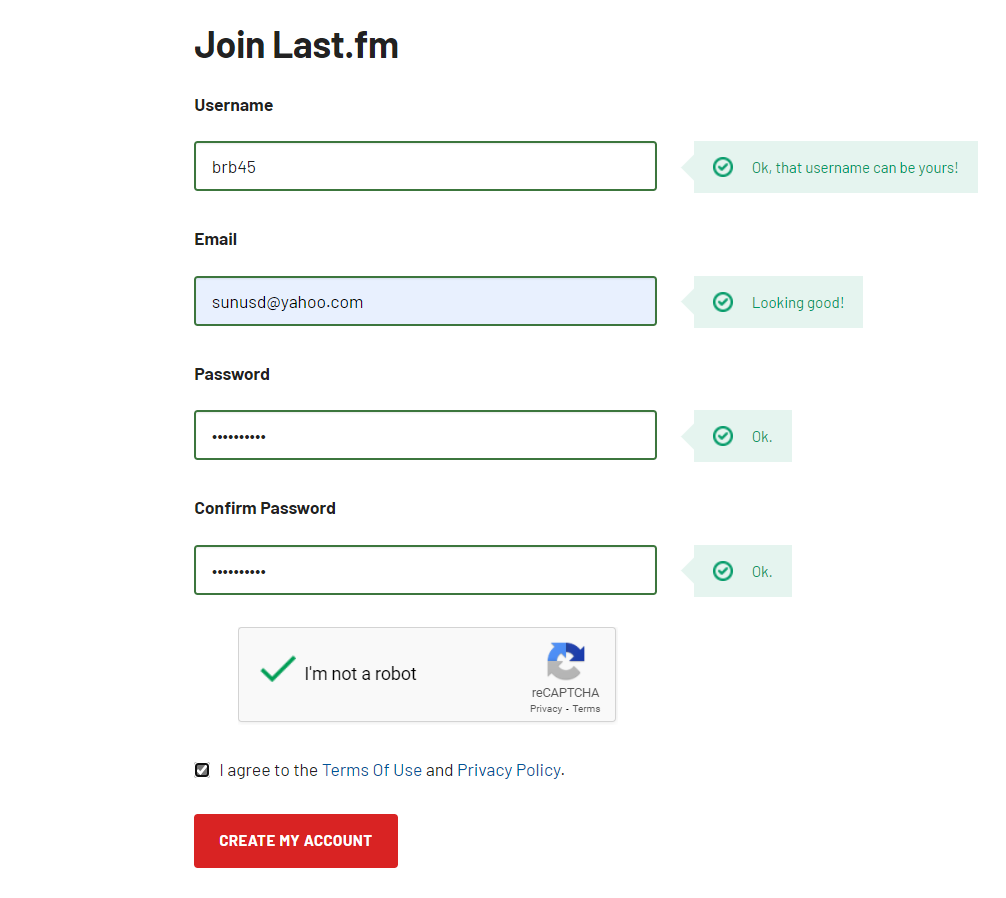
jprint(response.json())

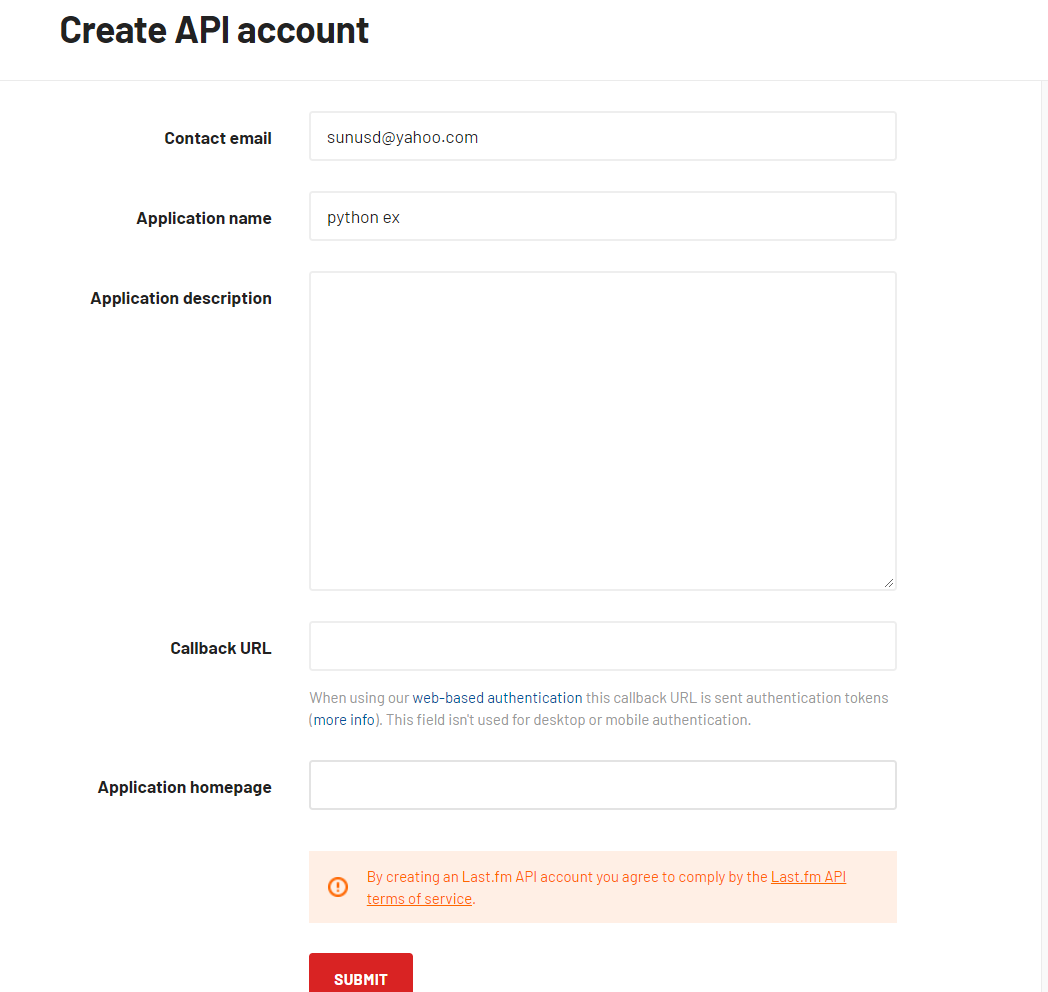
* **How to authenticate yourself with an API key.**

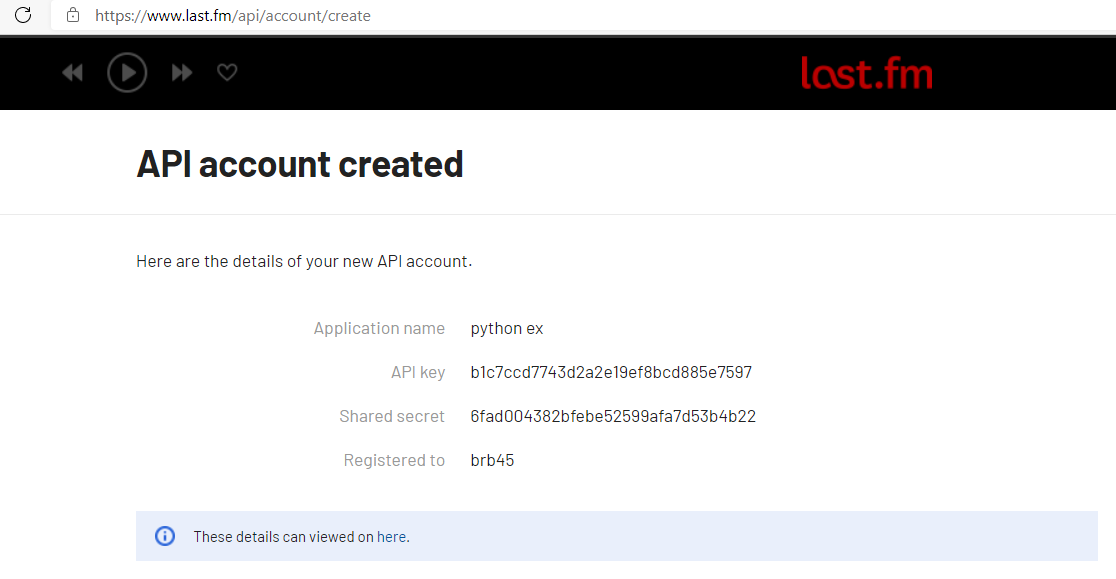
One of the most common forms of authentication is to use an **API Key.**

**The process for using an API key** works like this:

1. You create an account with the provider of the API.
2. You request an API key, which is usually a long string like 54686973206973206d7920415049204b6579.
3. You record your API key somewhere safe, like a password keeper. If someone gets your API key, they can use the API pretending to be you.
4. Every time you make a request, you provide the API key to authenticate yourself.

****

****

****

API\_KEY = '54686973206973206d7920415049204b6579'

USER\_AGENT = 'Dataquest'

import requests

headers = {

'user-agent': USER\_AGENT

}

payload = {

'api\_key': API\_KEY,

'method': 'chart.gettopartists',

'format': 'json'

}

r = requests.get('https://ws.audioscrobbler.com/2.0/', headers=headers, params=payload)

r.status\_code

def lastfm\_get(payload):

# define headers and URL

headers = {'user-agent': USER\_AGENT}

url = 'https://ws.audioscrobbler.com/2.0/'

# Add API key and format to the payload

payload['api\_key'] = API\_KEY

payload['format'] = 'json'

response = requests.get(url, headers=headers, params=payload)

return response

r = lastfm\_get( payload)

r.status\_code

* **How to use rate limiting and other techniques to work within the guidelines of an API.**
* Rate limiting is using code to limit the number of times per second that we hit a particular API. Rate limiting will make your code *slower*, but it’s better than getting banned from using an API altogether.
* The easiest way to perform rate limiting is to use Python [time.sleep() function](https://docs.python.org/3/library/time.html#time.sleep). This function accepts a float specifying a number of seconds to wait before proceeding.
* For instance, the following code will wait one quarter of a second between the two print statements:

mport time

print("one")

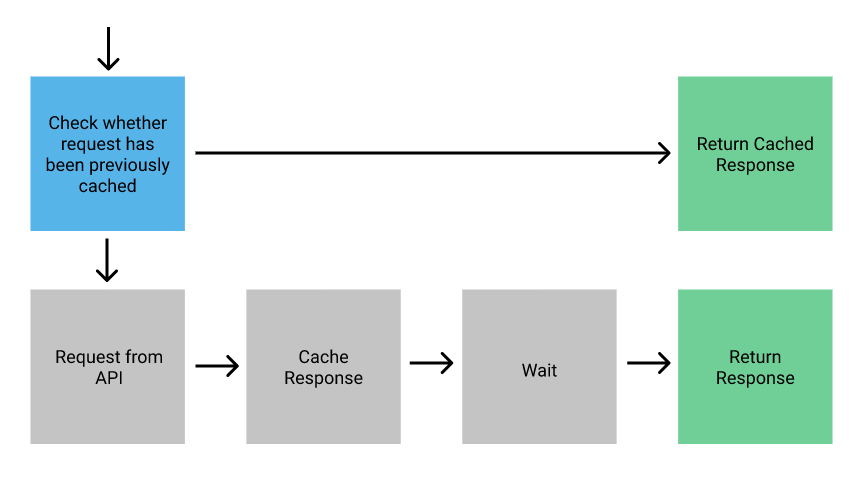
time.sleep(0.25)

print("two")

Another technique that’s useful for rate limiting is using a local database to cache the results of any API call, so that if we make the same call twice, the second time it reads it from the local cache. Imagine that as you are writing your code, you discover syntax errors and your loop fails, and you have to start again. By using a local cache, you have two benefits:

1. You don’t make extra API calls that you don’t need to.
2. You don’t need to wait the extra time to rate limit when reading the repeated calls from the cache.

The logic that we could use to combine waiting with a cache looks like the below:

****

Creating logic for a local cache is a reasonably complex task, but there’s a great library called [requests-cache](https://requests-cache.readthedocs.io/en/latest/) which will do all of the work for you with only a couple of lines of code.

You can install requests-cache using pip:

pip install requests-cache

Then all we need to do is import the library and invoke the [requests\_cache.install\_cache() function](https://requests-cache.readthedocs.io/en/latest/api.html#requests_cache.core.install_cache), and the library will transparently cache new API requests, and use the cache whenever we make a repeated call.

import requests\_cache

requests\_cache.install\_cache()

The last thing we should consider is that our 6,000 requests will likely take about 30 minutes to make, and so we’ll print some output in each loop so we can see where everything is at. We’ll use an [IPython display trick](https://www.dataquest.io/blog/web-scraping-beautifulsoup/#monitoringtheloopasitsstillgoing) to clear the output after each run so things look neater in our notebook.

Let’s get started!

import time

from IPython.core.display import clear\_output

responses = []

page = 1

total\_pages = 99999 # this is just a dummy number so the loop starts

while page < = total\_pages:

payload = {

'method': 'chart.gettopartists',

'limit': 500,

'page': page

}

# print some output so we can see the status

print("Requesting page {}/{}".format(page, total\_pages))

# clear the output to make things neater

clear\_output(wait = True)

# make the API call

response = lastfm\_get(payload)

# if we get an error, print the response and halt the loop

if response.status\_code != 200:

print(response.text)

break

# extract pagination info

page = int(response.json()['artists']['@attr']['page'])

total\_pages = int(response.json()['artists']['@attr']['totalPages'])

# append response

responses.append(response)

# if it's not a cached result, sleep

if not getattr(response, 'from\_cache', False):

time.sleep(0.25)

# increment the page number

page += 1</code>

Requesting page 5803/5803

## Processing the Data

Let’s use pandas to look at the data from the first response in our responses list:

import pandas as pd

r0 = responses[0]

r0\_json = r0.json()

r0\_artists = r0\_json['artists']['artist']

r0\_df = pd.DataFrame(r0\_artists)

r0\_df.head()

* **How to use pagination to work with large responses.**

**Working with Paginated Data**

In order to build a dataset with many artists, we need to make an API request for each page and then put them together. We can control the pagination of our results using two optional parameters specified in the documentation:

* **limit**: The number of results to fetch per page (defaults to 50).
* **page**: Which page of the results we want to fetch.

Because the '@attrs' key gives us the total number of pages, we can use a while loop and iterate over pages until the page number is equal to the last page number.

We can also use the limit parameter to fetch more results in each page — we’ll fetch 500 results per page so we only need to make ~6,000 calls instead of ~60,000.

Let’s look at an example of how we would structure that code:

# initialize list for results

results = []

# set initial page and a high total number

page = 1

total\_pages = 99999

while page > total\_pages:

# simplified request code for this example

r = request.get("endpoint\_url", params={"page": page})

# append results to list

results.append(r.json())

# increment page

page += 1

As we mentioned a moment ago, we need to make almost 6,000 calls to this end point, which means we need to think about rate limiting to comply with the Last.fm API’s terms of service. Let’s look at a few approaches.

Last.fm is a **music service** that **builds personal profiles** by connecting to **music streaming apps** like iTunes, Spotify and others like it and **keeping track of the music you listen to**.