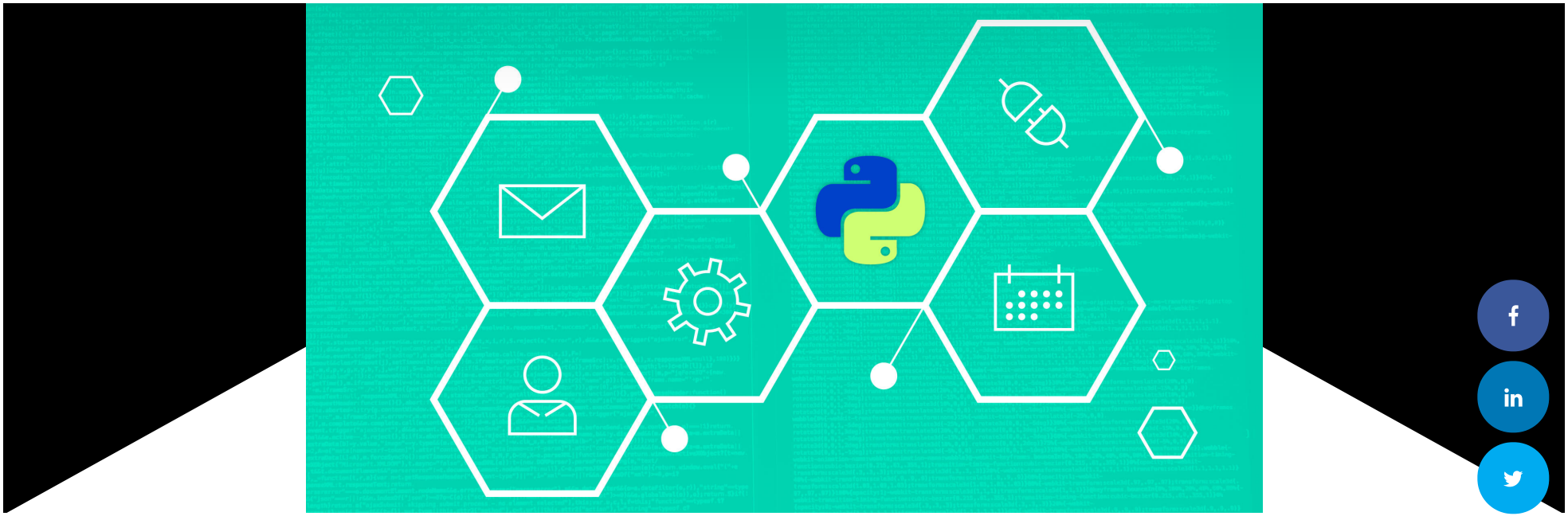


How to Use the Python Requests Module With REST APIs

Learn how to use the Python Requests module to interact with any REST API in the world.

The clear, simple syntax of Python makes it an ideal language to interact with REST APIs, and in typical Python fashion, there's a library made specifically to provide that functionality: Requests. Python Requests is a powerful tool that provides the simple elegance of Python to make HTTP requests to any API in the world. At Nylas, we built our REST APIs for email, calendar, and contacts on Python, and we process over 500 million API requests a day, so naturally, we depend a ton on the Python Requests library.

In this guide, we'll take a comprehensive look at making HTTP requests with Python Requests and learn how to use this functionality to integrate with REST APIs.

- How to handle HTTP errors with Python Requests
- How to make robust API Requests

The Roles of HTTP, APIs, and REST

An Application Programming Interface (API) is a web service that grants access to specific data and methods that other applications can access – and sometimes edit – via standard HTTP protocols, just like a website. This simplicity makes it easy to quickly integrate APIs into a wide variety of applications. REpresentational State Transfer (REST), is probably the most popular architectural style of APIs for web services. It consists of a set of guidelines designed to simplify client / server communication. REST APIs make data access much more straightforward and logical.

The Request

When you want to interact with data via a REST API, this is called a request. A request is made up of the following components:

Endpoint – The URL that delineates what data you are interacting with. Similar to how a web page URL is tied to a specific page, an endpoint URL is tied to a specific resource within an API.

Method – Specifies how you're interacting with the resource located at the provided endpoint. REST APIs can provide methods to enable full Create, Read, Update, and Delete (CRUD) functionality. Here are common methods most REST APIs provide:

Data – If you're using a method that involves changing data in a REST API, you'll need to include a data payload with the request that includes all data that will be created or modified.

Headers – Contain any metadata that needs to be included with the request, such as authentication tokens, the content type that should be returned, and any caching policies.

The Response

When you perform a request, you'll get a response from the API. Just like in the request, it'll have a response header and response data, if applicable. The response header consists of useful metadata about the response, while the response data returns what you actually requested. This can be any sort of data, as it's really dependent on the API. The text is usually returned as JSON, but other markdown languages like XML are also possible.

Let's look at a simple example of a request and a response. In the terminal, we'll use curl to make a GET request to the Open Notify API. This is a simple, yet nifty API that has information about astronauts that are currently in space:

```
curl -X GET "http://api.open-notify.org/astros.json"
```

You should see a response in JSON format that lists data about these astronauts, at the time of this article there are three people on a historic trip to the International Space Station:

```
{
  "number": 3,
```

```
    },
    {
      "craft": "ISS",
      "name": "Anatoly Ivanishin"
    },
    {
      "craft": "ISS",
      "name": "Ivan Vagner"
    }
  ]
}
```

How to Use Python Requests with REST APIs

Now, let's take a look at what it takes to integrate with a REST API using Python Requests. First, you'll need to have the necessary software; make sure you have Python and pip installed on your machine. Then, head over to the command line and install the python requests module with pip:

```
pip install requests
```

Now you're ready to start using Python Requests to interact with a REST API, make sure you import the Requests library into any scripts you want to use it in:

```
import requests
```

```
import requests
response = requests.get("http://api.open-notify.org/astros.json")
print(response)
>>>> Response<200>
```

The response object contains all the data sent from the server in response to your GET request, including headers and the data payload. When this code example prints the response object to the console it simply returns the name of the object's class and the status code the request returned (more on status codes later).

While this information might be useful, you're most likely interested in the content of the request itself, which can be accessed in a few ways:

```
response.content() # Return the raw bytes of the data payload
response.text() # Return a string representation of the data payload
response.json() # This method is convenient when the API returns JSON
```

How to Use Query Parameters

Queries can be used to filter the data that an API returns, and these are added as query parameters that are appended to the endpoint URL. With Python Requests, this is handled via the params argument, which accepts a dictionary object; let's see what that looks like when we use the Open Notify API to GET an estimate for when the ISS will fly over a specified point:

```
{
  'message': 'success',
  'request': {
    'altitude': 100,
    'datetime': 1590607799,
    'latitude': 45.0,
    'longitude': 180.0,
    'passes': 5
  },
  'response': [
    {'duration': 307, 'risetime': 1590632341},
    {'duration': 627, 'risetime': 1590637934},
    {'duration': 649, 'risetime': 1590643725},
    {'duration': 624, 'risetime': 1590649575},
    {'duration': 643, 'risetime': 1590655408}
  ]
}
```

How to Create and Modify Data With POST and PUT

In a similar manner as the query parameters, you can use the data argument to add the associated data for PUT and POST method requests.

```
# Create a new resource
```

How to Access REST Headers

You can also retrieve metadata from the response via headers. For example, to view the date of the response, just specify that with the `headers` property:

```
print(response.headers["date"])
>>>> 'Wed, 11 June 2020 19:32:24 GMT'
```

For open APIs, that covers the basics. However, many APIs can't be used by just anyone. For those, let's go over how to authenticate to REST APIs.

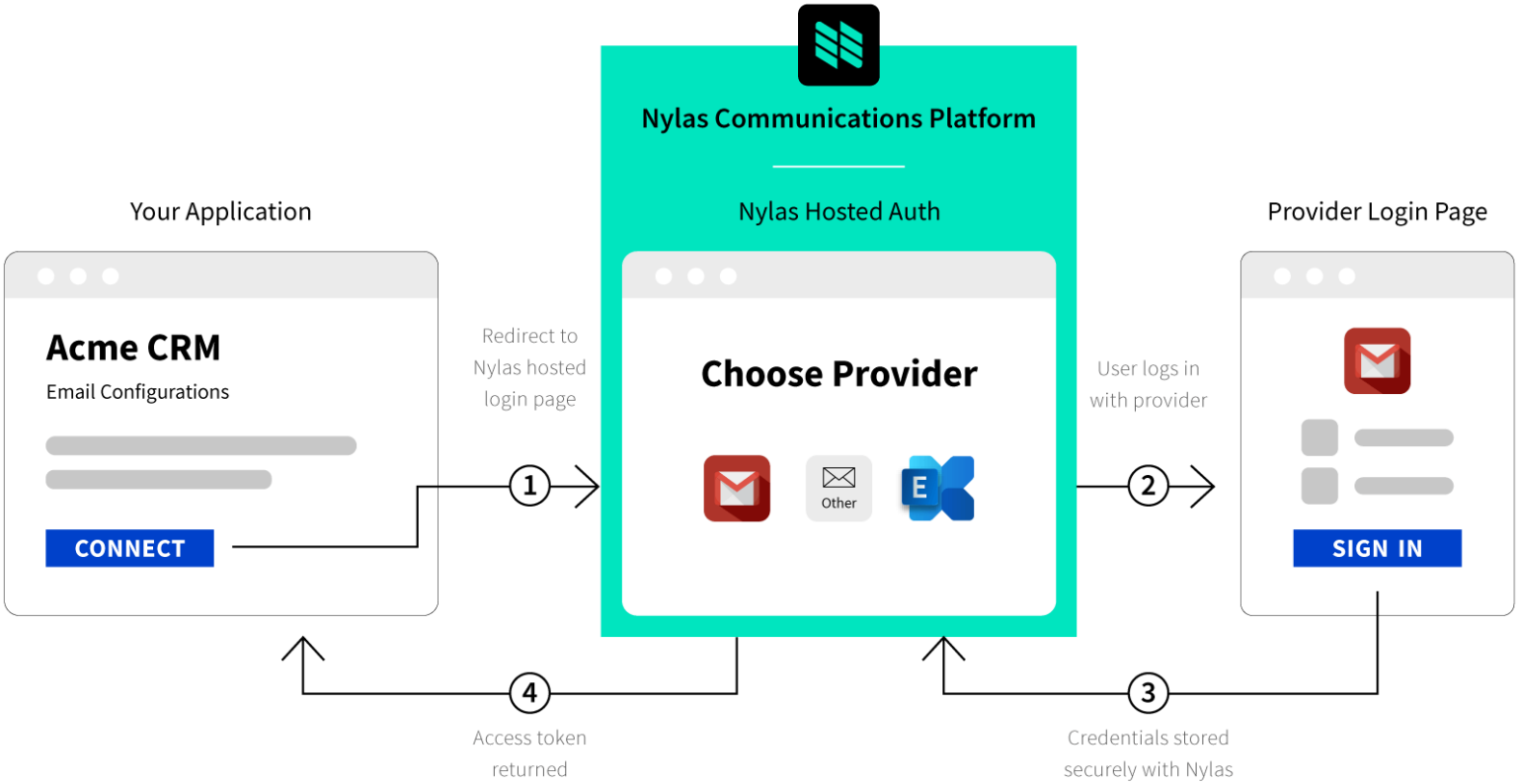
How to Authenticate to a REST API

So far you've seen how to interact with open REST APIs that don't require any authorization. However, many REST APIs require you to authenticate to them before you can access specific endpoints, particularly if they deal with sensitive data.

There are a few common authentication methods for REST APIs that can be handled with Python Requests. The simplest way is to pass your username and password to the appropriate endpoint as HTTP Basic Auth; this is equivalent to typing your username and password into a website.

```
requests.get(
  'https://api.github.com/user',
  auth=HTTPBasicAuth('username', 'password')
```

combination; the method to get an access token varies widely from API to API, but the most common framework for API authentication is OAuth. Here at Nylas, we use three-legged OAuth to grant an access token for user accounts that is restricted to scopes that define the specific data and functionality that can be accessed. This process is demonstrated in the Nylas Hosted Auth service.



There are quite a few other methods to authenticate to a REST API, including digest, Kerberos, NTLM, and AuthBase. The use of these depends on the architecture decisions of the REST API producer.

Use Sessions to Manage Access Tokens

Session objects come in handy when working with Python Requests as a tool to persist parameters that are needed for making multiple requests within a single session, like access tokens. Also, managing session cookies can provide a nice performance increase because you don't need to open a new connection for every request.

```
session = requests.Session()
session.headers.update({'Authorization': 'Bearer {access_token}'})
response = session.get('https://httpbin.org/headers')
```

How to Handle HTTP Errors With Python Requests

API calls don't always go as planned, and there's a multitude of reasons why API requests might fail that could be the fault of either the server or the client. If you're going to use a REST API, you need to understand how to handle the errors they output when things go wrong to make your code more robust. This section covers everything you need to know about handling HTTP errors with Python Requests.

The Basics of HTTP Status Codes

- 1xx Informational – Indicates that a request has been received and that the client should continue to make the requests for the data payload. You likely won't need to worry about these status codes while working with Python Requests.
- 2xx Successful – Indicates that a requested action has been received, understood, and accepted. You can use these codes to verify the existence of data before attempting to act on it.
- 3xx Redirection – Indicates that the client must make an additional action to complete the request like accessing the resource via a proxy or a different endpoint. You may need to make additional requests, or modify your requests to deal with these codes.
- 4xx Client Error – Indicates problems with the client, such as a lack of authorization, forbidden access, disallowed methods, or attempts to access nonexistent resources. This usually indicates configuration errors on the client application.
- 5xx Server Error – Indicates problems with the server that provides the API. There are a large variety of server errors and they often require the API provider to resolve.

How to Check for HTTP Errors With Python Requests

The response objects has a status_code attribute that can be used to check for any errors the API might have reported. The next example shows how to use this attribute to check for successful and 404 not found HTTP status codes, and you can use this same format for all HTTP status codes.

```
response = requests.get("http://api.open-notify.org/astros.json")
if (response.status_code == 200):
    print("The request was a success!")
```

To see this in action, try removing the last letter from the URL endpoint, the API should return a 404 status code.

If you want requests to raise an exception for all error codes (4xx and 5xx), you can use the raise_for_status() function and catch specific errors using Requests built-in exceptions. This next example accomplishes the same thing as the previous code example.

```
try:
    response = requests.get('http://api.open-notify.org/astros.json')
    response.raise_for_status()
    # Additional code will only run if the request is successful
except requests.exceptions.HTTPError as error:
    print(error)
    # This code will run if there is a 404 error.
```

TooManyRedirects

Something that is often indicated by 3xx HTTP status codes is the requirement to redirect to a different location for the resource you're requesting. This can sometimes result in a situation where you end up with an infinite redirect loop. The Python Requests module has the TooManyRedirects error that you can use to handle this problem. To resolve this problem, it's likely the URL you're using to access the resource is wrong and needs to be changed.

```
try:
```

You can optionally use the request options to set the maximum number of redirects:

```
response = requests.get('http://api.open-notify.org/astros.json',
max_redirects=2)
```

Or disable redirecting completely within your request options:

```
response = requests.get('http://api.open-notify.org/astros.json',
allow_redirects=False)
```

ConnectionError

So far, we've only looked at errors that come from an active server. What happens if you don't receive a response from the server at all? Connection errors can occur for many different reasons, including a DNS failure, refused connection, internet connectivity issues or latency somewhere in the network. Python Requests offers the ConnectionError exception that indicates when your client is unable to connect to the server.

```
try:
    response = requests.get('http://api.open-notify.org/astros.json')
    # Code here will only run if the request is successful
except requests.ConnectionError as error:
    print(error)
```

Timeout

Timeout errors occur when you're able to connect to the API server, but it doesn't complete the request within the allotted amount of time. Similar to the other errors we've looked at, Python Requests can handle this error with a Timeout exception:

```
try:
    response = requests.get('http://api.open-notify.org/astros.json',
timeout=0.00001)
    # Code here will only run if the request is successful
except requests.Timeout as error:
    print(error)
```

In this example, the timeout was set as a fraction of a second via the request options. Most APIs are unable to respond this quickly, so the code will produce a timeout exception. You can avoid this error by setting longer timeouts for your script, optimizing your requests to be smaller, or setting up a retry loop for the request. This can also sometimes indicate a problem with the API provider. One final solution is to incorporate asynchronous API calls to prevent your code from stopping while it waits for larger responses.

How to Make Robust API Requests

As we've seen, the Requests module elegantly handles common API request errors by utilizing exception handling in Python. If we put all of the errors we've talked about together, we have a rather seamless way

```
    # Code here will only run if the request is successful
except requests.exceptions.HTTPError as errh:
    print(errh)
except requests.exceptions.ConnectionError as errc:
    print(errc)
except requests.exceptions.Timeout as errt:
    print(errt)
except requests.exceptions.RequestException as err:
    print(err)
```

Learn More About Python

If you've made it this far, congrats! You're well on your way to becoming a Python Requests wizard for whom no REST API is too great a match. Want to keep learning? We have tons of knowledgable Python experts here at Nylas, and we have in-depth content on our blog about packaging and deploying Python code to production, and using environment variables to make your Python code more secure.

You may also like:

Verification Process and Security Assessment for Developers

Mail

Google People API vs. Contacts API: What You Should Know About Migrating

Outlook Calendar

Gmail Contacts

Exchange Contacts

Office 365 Contacts

Outlook Contacts