Universal Client Documentation

Release 0.1

David Greisen

Sep 27, 2017

Contents

Python 2 and Python 3 Compatible

Docs: https://universal-client.readthedocs.org
Repository: https://github.com/dgreisen/universalclient
PyPI: https://pypi.python.org/pypi/UniversalClient

Universal Client is an Apache 2 licensed client for quickly and easily interacting with any REST api. Universal Client is compatible with Python 2 and Python 3.

You want to use a cool api, but the python client doesn't exist, or is out of date, or doesn't support the feature you want to use. So you start making raw HTTP requests. But, even with the best http library, passing in the right URL and options for every call is error prone and quickly becomes cumbersome for anything more than trivial scenarios.

The Universal Client is a wrapper around the excellent Requests HTTP library. A Client instance corresponds to a single endpoint in the api. Use dot notation to navigate the api paths. You can set any Requests parameters on a client instance and all child instances will inherit those parameters.

A quick example:

```
>>> from universalclient import Client
# create a client pointing to google
>>> google = Client("http://google.com")
# set a google cookie so we get "personalized" results (Yay!)
>>> google = google.cookies(gv="5d41402abc4b2a76b9719d911017c592")
# get the google home page at http://google.com
>>> resp = google.get()
# create an end point to google images, which inherits the cookies
>>> images = google.images
# get google site at http://google.com/images
>>> resp = images.get()
```

The Client stores a complete api call. It is immutable - any modification returns a new client. This allows you to reuse endpoints over and over.

Contents:

CHAPTER 1

Basics

Creating a URL

UniversalClient is a wrapper around the excellent Requests library for making HTTP requests. Lets create a client for a fictitious photo sharing service:

```
>>> from universalclient import Client
# create a client pointing to myImages.com
>>> Client("http://myImages.com")
get: http://myImages.com
```

A client object is returned. In this case, the client object points to the URL "http://myImages.com" and has defaulted to the "get" HTTP method. We can add to this base URL using dot notation. Getting an attribute on the client will return a new Client with the name of that attribute appended to the initial Client's URL:

```
>>> Client("http://myImages.com").images
get: http://myImages.com/user
```

Clients are immutable:

```
>>> root = Client("http://myImages.com").images
>>> davidsImages = root.user.dgreisen
>>> root
get: http://myImages.com
>>> davidsImages
get: http://myImages.com/user/dgreisen
```

Making requests

You can make the default request by calling client.request(), which returns a Requests response object. You can make a specific type of request by calling that function:

```
>>>
# make the default get request
>>> davidsImages.request()
<Response [200]>
# make a head request
>>> davidsImages.head()
<Response [200]>
```

Arguments

Any keyword argument available for Requests.request can be set on your client. You can do this when a client is created, any time after creation, or just before sending the request. Any kwargs passed to Client(), client.request(), client.get(), or any other client.<HTTP-METHOD>(), will be passed directly to request.request(),

At creation:

```
>>> root = Client("http://myImages.com", auth=('user', 'pass'), method="get", params={
↪"api_key":"123"}, headers={'Content-Type': 'application/json', 'Authorization':
↪'Bearer <api_key>'})
```

Updating any client after initial creation:

```
>>> uploadImage = root.user.dgreisen.images.method("put").params(size="full")
```

Just before sending the request:

```
>>> uploadImage.request(files={'file': open('birthdayBash.jpg', 'rb'))
<Response [200]>
```

You can display all the arguments:

```
>>> uploadImage.getArgs()
{'_http': <module 'requests' from '/usr/local/lib/python2.7/dist-packages/requests/__
↪init__.pyc'>, 'params': {'api_key': '123', 'size': 'full'}, 'method': 'put', 'auth
↪': ('user', 'pass'), '_path': ['http://myImages.com', 'user', 'dgreisen', 'images']}
```

You will notice there are a couple of other keys/values in there starting with an underscore. Underscore keys are not passed to requests.request(). You can read about some of them in the *Advanced* section. You will also notice that the params object was updated with the new keys rather than being replaced by the new keys. Args that are dictionaries are updated rather than replaced.

If you set an argument that already has a value, it will be overridden. You can also delete an existing value:

```
>>> getAllImages = uploadImage.method()

>>> "method" in getAllImages.getArgs()
False
```

No error will be thrown if an argument to be deleted does not exist.

CHAPTER 2

Advanced

Creating a URL

The URL we created in *Basics* section:

```
>>> root = Client("http://myImages.com").images
>>> davidsImages = root.user.dgreisen
```

is great if we know we are going to do a bunch of stuff to davidsImages. But what if we want to be able to pass in a userid that isn't known until runtime? That's when the _ (underscore) method is used. The argument to _ will be added to the URL, just as if it were appended with dot notation:

```
>>> somebody = "jsmith"
>>> somebodysImages = root.user._(somebody)
>>> somebodysImages
get: http://myImages.com/images/user/jsmith
```

Even better, the string passed to _ can be formatted with all the args passed to a request() (or get(), post(), etc.). Thus, we can make an endpoint for adding an arbitrary image to an arbitrary users album:

```
>>> addImage = root.user._({}).images._({1}).method("put")
>>> addImage.request("dgreisen", "birthdayBash", files={'file': open('birthdayBash.jpg
↪', 'rb'))
<Response [200]>
>>> addImage.request("jsmith", "birthdayBash", files={'file': open('birthdayBash.jpg',
↪ 'rb'))
<Response [200]>
```

You can only use positional arguments to format the string since the keyword arguments are passed to requests.request(). All arguments are passed to each string, so if you have multiple strings appended with _, you must use positional args properly.

Data formatting

Sometimes the data you need to send to a specific endpoint has to be formatted in a specific way. You can perform this manipulation by adding a dataFilter. A dataFilter is a function that takes one argument, the data, and returns a single value, the data to be sent to the server. Data will be transformed just before it is sent in the request.

UniversalClient currently comes with one dataFilter built in, jsonFilter, which encodes the data as json:

```
>>> from universalclient import jsonFilter
>>> response = root.user.jdoe.put(data={"name": "Jane Doe", "email": "jdoe@example.com
↪"}, dataFilter=jsonFilter)
>>> response.request.body
'{"name": "Jane Doe", "email": "jdoe@example.com"}'
```

If you write a dataFilter that you think others would find useful, please submit a pull request.

Oauth

Because UniversalClient is just Requests at heart, you can use Rauth for oauth authentication. To use, create a fully authenticated Rauth client (see Rauth documentation). Then pass the fully authenticated client into your universal client:

```
>>> root = Client("http://myImages.com").oauth(rauth_client).images
```

The client then uses the rauth client to make requests, rather than a default Requests instance. Please note, this functionality has not been tested. Tests and bug reports are welcome.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search