requests_toolbelt Documentation

Release 0.4.0

Ian Cordasco, Cory Benfield

April 03, 2015

Contents

This is a collection of utilities that some users of python-requests might need but do not belong in requests proper. The library is actively maintained by members of the requests core development team, and so reflects the functionality most requested by users of the requests library.

To get an overview of what the library contains, consult the *user* documentation.

CHAPTER 1

Overview

1.1 requests toolbelt

This is just a collection of utilities for python-requests, but don't really belong in `requests` proper. The minimum tested requests version is `2.1.0`. In reality, the toolbelt should work with `2.0.1` as well, but some idiosyncracies prevent effective or sane testing on that version.

1.1.1 multipart/form-data Encoder

The main attraction is a streaming multipart form-data object, `MultipartEncoder`. Its API looks like this:

```
from requests_toolbelt import MultipartEncoder
import requests

m = MultipartEncoder(
    fields={'field0': 'value', 'field1': 'value',
            'field2': ('filename', open('file.py', 'rb'), 'text/plain')}
    )

r = requests.post('http://httpbin.org/post', data=m,
                  headers={'Content-Type': m.content_type})
```

You can also use `multipart/form-data` encoding for requests that don't require files:

```
from requests_toolbelt import MultipartEncoder
import requests

m = MultipartEncoder(fields={'field0': 'value', 'field1': 'value'})

r = requests.post('http://httpbin.org/post', data=m,
                  headers={'Content-Type': m.content_type})
```

Or, you can just create the string and examine the data:

```
# Assuming `m` is one of the above
m.to_string()  # Always returns unicode
```

1.1.2 User-Agent constructor

You can easily construct a requests-style `User-Agent` string:

```
from requests_toolbelt import user_agent

headers = {
    'User-Agent': user_agent('my_package', '0.0.1')
    }

r = requests.get('https://api.github.com/users', headers=headers)
```

1.1.3 SSLAdapter

The `SSLAdapter` was originally published on Cory Benfield's blog. This adapter allows the user to choose one of the SSL protocols made available in Python's `ssl` module for outgoing HTTPS connections:

```
from requests_toolbelt import SSLAdapter
import requests
import ssl

s = requests.Session()
s.mount('https://', SSLAdapter(ssl.PROTOCOL_TLSv1))
```

1.1.4 Known Issues

On Python 3.3.0 and 3.3.1, the standard library's `http` module will fail when passing an instance of the `MultipartEncoder`. This is fixed in later minor releases of Python 3.3. Please consider upgrading to a later minor version or Python 3.4. *There is absolutely nothing this library can do to work around that bug.*

CHAPTER 2

Full Documentation

2.1 Transport Adapters

The toolbelt comes with several different transport adapters for you to use with requests. The transport adapters are all kept in `requests_toolbelt.adapters` and include

- `requests_toolbelt.adapters.fingerprint.FingerprintAdapter`
- `requests_toolbelt.adapters.socket_options.SocketOptionsAdapter`
- `requests_toolbelt.adapters.socket_options.TCPKeepAliveAdapter`
- `requests_toolbelt.adapters.source.SourceAddressAdapter`
- `requests_toolbelt.adapters.ssl.SSLAdapter`

2.1.1 FingerprintAdapter

New in version 0.4.0.

By default, requests will validate a server's certificate to ensure a connection is secure. In addition to this, the user can provide a fingerprint of the certificate they're expecting to receive. Unfortunately, the requests API does not support this fairly rare use-case. When a user needs this extra validation, they should use the `FingerprintAdapter` class to perform the validation.

class `requests_toolbelt.adapters.fingerprint.`**`FingerprintAdapter`**(*fingerprint*, ***kwargs*)

A HTTPS Adapter for Python Requests that verifies certificate fingerprints, instead of certificate hostnames.

Example usage:

```
import requests
import ssl
from requests_toolbelt.adapters.fingerprint import FingerprintAdapter

twitter_fingerprint = '...'
s = requests.Session()
s.mount(
    'https://twitter.com',
    FingerprintAdapter(twitter_fingerprint)
)
```

The fingerprint should be provided as a hexadecimal string, optionally containing colons.

2.1.2 SSLAdapter

The `SSLAdapter` is the canonical implementation of the adapter proposed on Cory Benfield's blog, here. This adapter allows the user to choose one of the SSL/TLS protocols made available in Python's `ssl` module for outgoing HTTPS connections.

In principle, this shouldn't be necessary: compliant SSL servers should be able to negotiate the required SSL version. In practice there have been bugs in some versions of OpenSSL that mean that this negotiation doesn't go as planned. It can be useful to be able to simply plug in a Transport Adapter that can paste over the problem.

For example, suppose you're having difficulty with the server that provides TLS for GitHub. You can work around it by using the following code:

```
from requests_toolbelt.adapters.ssl import SSLAdapter

import requests
import ssl

s = requests.Session()
s.mount('https://github.com/', SSLAdapter(ssl.PROTOCOL_TLSv1))
```

Any future requests to GitHub made through that adapter will automatically attempt to negotiate TLSv1, and hopefully will succeed.

class `requests_toolbelt.adapters.ssl.`**`SSLAdapter`**(*ssl_version=None*, ***kwargs*)

A HTTPS Adapter for Python Requests that allows the choice of the SSL/TLS version negotiated by Requests. This can be used either to enforce the choice of high-security TLS versions (where supported), or to work around misbehaving servers that fail to correctly negotiate the default TLS version being offered.

Example usage:

```
>>> import requests
>>> import ssl
>>> from requests_toolbelt import SSLAdapter
>>> s = requests.Session()
>>> s.mount('https://', SSLAdapter(ssl.PROTOCOL_TLSv1))
```

You can replace the chosen protocol with any that are available in the default Python SSL module. All subsequent requests that match the adapter prefix will use the chosen SSL version instead of the default.

2.1.3 SourceAddressAdapter

New in version 0.3.0.

The `SourceAddressAdapter` allows a user to specify a source address for their connnection.

class `requests_toolbelt.adapters.source.`**`SourceAddressAdapter`**(*source_address*, ***kwargs*)

A Source Address Adapter for Python Requests that enables you to choose the local address to bind to. This allows you to send your HTTP requests from a specific interface and IP address.

Example usage:

```
import requests
from requests_toolbelt.adapters.source import SourceAddressAdapter

s = requests.Session()
s.mount('http://', SourceAddressAdapter('10.10.10.10'))
```

2.1.4 SocketOptionsAdapter

New in version 0.4.0.

Note: This adapter will only work with requests 2.4.0 or newer. The ability to set arbitrary socket options does not exist prior to requests 2.4.0.

The `SocketOptionsAdapter` allows a user to pass specific options to be set on created sockets when constructing the Adapter without subclassing. The adapter takes advantage of `urllib3`'s support for setting arbitrary socket options for each `urllib3.connection.HTTPConnection` (and `HTTPSConnection`).

To pass socket options, you need to send a list of three-item tuples. For example, `requests` and `urllib3` disable Nagle's Algorithm by default. If you need to re-enable it, you would do the following:

```
import socket
import requests
from requests_toolbelt.adapters.socket_options import SocketOptionsAdapter

nagles = [(socket.IPPROTO_TCP, socket.TCP_NODELAY, 0)]
session = requests.Session()
for scheme in session.adapters.keys():
    session.mount(scheme, SocketOptionsAdapter(socket_options=nagles))
```

This would re-enable Nagle's Algorithm for all `http://` and `https://` connections made with that session.

class `requests_toolbelt.adapters.socket_options.`**`SocketOptionsAdapter`**(***kwargs*)

An adapter for requests that allows users to specify socket options.

Since version 2.4.0 of requests, it is possible to specify a custom list of socket options that need to be set before establishing the connection.

Example usage:

```
>>> import socket
>>> import requests
>>> from requests_toolbelt.adapters import socket_options
>>> s = requests.Session()
>>> opts = [(socket.IPPROTO_TCP, socket.TCP_NODELAY, 0)]
>>> adapter = socket_options.SocketOptionsAdapter(socket_options=opts)
>>> s.mount('http://', adapter)
```

You can also take advantage of the list of default options on this class to keep using the original options in addition to your custom options. In that case, `opts` might look like:

```
>>> opts = socket_options.SocketOptionsAdapter.default_options + opts
```

2.1.5 TCPKeepAliveAdapter

New in version 0.4.0.

Note: This adapter will only work with requests 2.4.0 or newer. The ability to set arbitrary socket options does not exist prior to requests 2.4.0.

The `TCPKeepAliveAdapter` allows a user to pass specific keep-alive related options as keyword parameters as well as arbitrary socket options.

Note: Different keep-alive related socket options may not be available for your platform. Check the socket module

for the availability of the following constants:

- `socket.TCP_KEEPIDLE`
- `socket.TCP_KEEPCNT`
- `socket.TCP_KEEPINTVL`

The adapter will silently ignore any option passed for a non-existent option.

An example usage of the adapter:

```
import requests
from requests_toolbelt.adapter.socket_options import TCPKeepAliveAdapter

session = requests.Session()
keep_alive = TCPKeepAliveAdapter(idle=120, count=20, interval=30)
session.mount('https://region-a.geo-1.compute.hpcloudsvc.com', keep_alive)
session.post('https://region-a.geo-1.compute.hpcloudsvc.com/v2/1234abcdef/servers',
             # ...
             )
```

In this case we know that creating a server on HP Public Cloud can cause requests to hang without using TCP Keep-Alive. So we mount the adapter specifically for that domain, instead of adding it to every `https://` and `http://` request.

class `requests_toolbelt.adapters.socket_options.`**`TCPKeepAliveAdapter`**`(`*`**kwargs`*`)`

An adapter for requests that turns on TCP Keep-Alive by default.

The adapter sets 4 socket options:

- `SOL_SOCKET` `SO_KEEPALIVE` - This turns on TCP Keep-Alive
- `IPPROTO_TCP` `TCP_KEEPINTVL` 20 - Sets the keep alive interval
- `IPPROTO_TCP` `TCP_KEEPCNT` 5 - Sets the number of keep alive probes
- `IPPROTO_TCP` `TCP_KEEPIDLE` 60 - Sets the keep alive time if the socket library has the `TCP_KEEPIDLE` constant

The latter three can be overridden by keyword arguments (respectively):

- `idle`
- `interval`
- `count`

You can use this adapter like so:

```
>>> from requests_toolbelt.adapters import socket_options
>>> tcp = socket_options.TCPKeepAliveAdapter(idle=120, interval=10)
>>> s = requests.Session()
>>> s.mount('http://', tcp)
```

2.2 Authentication

requests supports Basic Authentication and HTTP Digest Authentication by default. There are also a number of third-party libraries for authentication with:

- OAuth

- NTLM
- Kerberos

The `requests_toolbelt.auth` provides extra authentication features in addition to those. It provides the following authentication classes:

- `requests_toolbelt.auth.guess.GuessAuth`
- `requests_toolbelt.auth.http_proxy_digest.HTTPProxyDigestAuth`
- `requests_toolbelt.auth.handler.AuthHandler`

2.2.1 AuthHandler

The `AuthHandler` is a way of using a single session with multiple websites that require authentication. If you know what websites require a certain kind of authentication and what your credentials are.

Take for example a session that needs to authenticate to GitHub's API and GitLab's API, you would set up and use your `AuthHandler` like so:

```
import requests
from requests_toolbelt.auth.handler import AuthHandler

def gitlab_auth(request):
    request.headers['PRIVATE-TOKEN'] = 'asecrettoken'

handler = AuthHandler({
    'https://api.github.com': ('sigmavirus24', 'apassword'),
    'https://gitlab.com': gitlab_auth,
})

session = requests.Session()
session.auth = handler
r = session.get('https://api.github.com/user')
# assert r.ok
r2 = session.get('https://gitlab.com/api/v3/projects')
# assert r2.ok
```

Note: You **must** provide both the scheme and domain for authentication. The `AuthHandler` class will check both the scheme and host to ensure your data is not accidentally exposed.

---

class `requests_toolbelt.auth.handler.`**`AuthHandler`**(*strategies*)

The `AuthHandler` object takes a dictionary of domains paired with authentication strategies and will use this to determine which credentials to use when making a request. For example, you could do the following:

```
from requests import HTTPDigestAuth
from requests_toolbelt.auth.handler import AuthHandler

import requests

auth = AuthHandler({
    'https://api.github.com': ('sigmavirus24', 'fakepassword'),
    'https://example.com': HTTPDigestAuth('username', 'password')
})

r = requests.get('https://api.github.com/user', auth=auth)
# => <Response [200]>
r = requests.get('https://example.com/some/path', auth=auth)
```

```
# => <Response [200]>

s = requests.Session()
s.auth = auth
r = s.get('https://api.github.com/user')
# => <Response [200]>
```

Warning: `requests.auth.HTTPDigestAuth` is not yet thread-safe. If you use `AuthHandler` across multiple threads you should instantiate a new AuthHandler for each thread with a new HTTPDigestAuth instance for each thread.

add_strategy(*domain*, *strategy*)
Add a new domain and authentication strategy.

Parameters

- **domain** (*str*) – The domain you wish to match against. For example: `'https://api.github.com'`
- **strategy** (*str*) – The authentication strategy you wish to use for that domain. For example: `('username', 'password')` or `requests.HTTPDigestAuth('username', 'password')`

```
a = AuthHandler({})
a.add_strategy('https://api.github.com', ('username', 'password'))
```

get_strategy_for(*url*)
Retrieve the authentication strategy for a specified URL.

Parameters **url** (*str*) – The full URL you will be making a request against. For example, `'https://api.github.com/user'`

Returns Callable that adds authentication to a request.

```
import requests
a = AuthHandler({'example.com', ('foo', 'bar')})
strategy = a.get_strategy_for('http://example.com/example')
assert isinstance(strategy, requests.auth.HTTPBasicAuth)
```

remove_strategy(*domain*)
Remove the domain and strategy from the collection of strategies.

Parameters **domain** (*str*) – The domain you wish remove. For example, `'https://api.github.com'`.

```
a = AuthHandler({'example.com', ('foo', 'bar')})
a.remove_strategy('example.com')
assert a.strategies == {}
```

2.2.2 GuessAuth

The `GuessAuth` authentication class automatically detects whether to use basic auth or digest auth:

```
import requests
from requests_toolbelt.auth import GuessAuth

requests.get('http://httpbin.org/basic-auth/user/passwd',
             auth=GuessAuth('user', 'passwd'))
```

```
requests.get('http://httpbin.org/digest-auth/auth/user/passwd',
             auth=GuessAuth('user', 'passwd'))
```

Detection of the auth type is done via the `WWW-Authenticate` header sent by the server. This requires an additional request in case of basic auth, as usually basic auth is sent preemptively. If the server didn't explicitly require authentication, no credentials are sent.

class `requests_toolbelt.auth.guess.`**`GuessAuth`**(*username*, *password*)
Guesses the auth type by the WWW-Authentication header.

2.2.3 HTTPProxyDigestAuth

The `HTTPProxyDigestAuth` use digest authentication between the client and the proxy.

```
import requests
from requests_toolbelt.auth.http_proxy_digest import HTTPProxyDigestAuth


proxies = {
    "http": "http://PROXYSERVER:PROXYPORT",
    "https": "https://PROXYSERVER:PROXYPORT",
}
url = "https://toolbelt.readthedocs.org/"
auth = HTTPProxyDigestAuth("USERNAME", "PASSWORD")
requests.get(url, proxies=proxies, auth=auth)
```

Program would raise error if the username or password is rejected by the proxy.

class `requests_toolbelt.auth.http_proxy_digest.`**`HTTPProxyDigestAuth`**(**args*, ***kwargs*)
HTTP digest authentication between proxy

Parameters **stale_rejects** (*int*) – The number of rejects indicate that: the client may wish to simply retry the request with a new encrypted response, without reprompting the user for a new username and password. i.e., retry build_digest_header

2.3 Deprecated Requests Utilities

Requests has decided to deprecate some utility functions in `requests.utils`. To ease users' lives, they've been moved to `requests_toolbelt.utils.deprecated`. A collection of functions deprecated in requests.utils.

`requests_toolbelt.utils.deprecated.`**`get_encodings_from_content`**(*content*)
Return encodings from given content string.

```
import requests
from requests_toolbelt.utils import deprecated

r = requests.get(url)
encodings = deprecated.get_encodings_from_content(r)
```

Parameters **content** (*bytes*) – bytestring to extract encodings from.

`requests_toolbelt.utils.deprecated.`**`get_unicode_from_response`**(*response*)
Return the requested content back in unicode.

This will first attempt to retrieve the encoding from the response headers. If that fails, it will use `requests_toolbelt.utils.deprecated.get_encodings_from_content()` to determine encodings from HTML elements.

```
import requests
from requests_toolbelt.utils import deprecated

r = requests.get(url)
text = deprecated.get_unicode_from_response(r)
```

Parameters **response** (*requests.models.Response*) – Response object to get unicode content from.

2.4 Utilities for Downloading Streaming Reponses

`requests_toolbelt.downloadutils.stream.`**`stream_response_to_file`**(*response*, *path=None*)

Stream a response body to the specified file.

Either use the `path` provided or use the name provided in the `Content-Disposition` header.

Warning: If you pass this function an open file-like object as the `path` parameter, the function will not close that file for you.

Warning: This function will not automatically close the response object passed in as the `response` parameter.

If no `path` parameter is supplied, this function will parse the `Content-Disposition` header on the response to determine the name of the file as reported by the server.

```
import requests
from requests_toolbelt import exceptions
from requests_toolbelt.downloadutils import stream

r = requests.get(url, stream=True)
try:
    filename = stream.stream_response_to_file(r)
except exceptions.StreamingError as e:
    # The toolbelt could not find the filename in the
    # Content-Disposition
    print(e.message)
```

You can also specify the filename as a string. This will be passed to the built-in `open()` and we will read the content into the file.

```
import requests
from requests_toolbelt.downloadutils import stream

r = requests.get(url, stream=True)
filename = stream.stream_response_to_file(r, path='myfile')
```

Instead, if you want to manage the file object yourself, you need to provide either a `io.BytesIO` object or a file opened with the *'b'* flag. See the two examples below for more details.

```
import requests
from requests_toolbelt.downloadutils import stream
```

```
with open('myfile', 'wb') as fd:
    r = requests.get(url, stream=True)
    filename = stream.stream_response_to_file(r, path=fd)

print('{0} saved to {1}'.format(url, filename))

import io
import requests
from requests_toolbelt.downloadutils import stream

b = io.BytesIO()
r = requests.get(url, stream=True)
filename = stream.stream_response_to_file(r, path=b)
assert filename is None
```

Parameters

- **response** (*requests.models.Response*) – A Response object from requests
- **path** (`str`, or object with a `write()`) – *(optional)*, Either a string with the path to the location to save the response content, or a file-like object expecting bytes.

Returns The name of the file, if one can be determined, else None

Return type str

Raises `requests_toolbelt.exceptions.StreamingError`

2.5 Custom Toolbelt Exceptions

Below are the exception classes used by the toolbelt to provide error details to the user of the toolbelt. Collection of exceptions raised by requests-toolbelt.

exception `requests_toolbelt.exceptions.`**`StreamingError`**
Used in `requests_toolbelt.downloadutils.stream`.

2.6 Using requests with Threading

New in version 0.4.0.

The toolbelt provides a simple API for using requests with threading.

A requests Session is documented as threadsafe but there are still a couple corner cases where it isn't perfectly thread-safe. The best way to use a Session is to use one per thread.

The implementation provided by the toolbelt is naïve. This means that we use one session per thread and we make no effort to synchronize attributes (e.g., authentication, cookies, etc.). It also means that we make no attempt to direct a request to a session that has already handled a request to the same domain. In other words, if you're making requests to multiple domains, the toolbelt's Pool will not try to send requests to the same domain to the same thread.

This module provides three classes:

- `Pool`
- `ThreadResponse`
- `ThreadException`

In 98% of the situations you'll want to just use a `Pool` and you'll treat a `ThreadResponse` as if it were a regular `requests.Response`.

Here's an example:

```
# This example assumes Python 3
import queue
from requests_toolbelt.threaded import pool

jobs = queue.Queue()
urls = [
    # My list of URLs to get
]

for url in urls:
    queue.put({'method': 'GET', 'url': url})

p = pool.Pool(job_queue=q)
p.join_all()

for response in p.responses():
    print('GET {0}. Returned {1}.'.format(response.request_kwargs['url'],
                                          response.status_code))
```

This is clearly a bit underwhelming. This is why there's a short-cut class method to create a `Pool` from a list of URLs.

```
from requests_toolbelt.threaded import pool

urls = [
    # My list of URLs to get
]

p = pool.Pool.from_urls(urls)
p.join_all()

for response in p.responses():
    print('GET {0}. Returned {1}.'.format(response.request_kwargs['url'],
                                          response.status_code))
```

If one of the URLs in your list throws an exception, it will be accessible from the `exceptions()` generator.

```
from requests_toolbelt.threaded import pool

urls = [
    # My list of URLs to get
]

p = pool.Pool.from_urls(urls)
p.join_all()

for exc in p.exceptions():
    print('GET {0}. Raised {1}.'.format(exc.request_kwargs['url'],
                                        exc.message))
```

If instead, you want to retry the exceptions that have been raised you can do the following:

```
from requests_toolbelt.threaded import pool

urls = [
    # My list of URLs to get
```

```
]

p = pool.Pool.from_urls(urls)
p.join_all()

new_pool = pool.Pool.from_exceptions(p.exceptions())
new_pool.join_all()
```

Not all requests are advisable to retry without checking if they should be retried. You would normally check if you want to retry it.

The `Pool` object takes 4 other keyword arguments:

- `initializer`

  This is a callback that will initialize things on every session created. The callback must return the session.

- `auth_generator`

  This is a callback that is called *after* the initializer callback has modified the session. This callback must also return the session.

- `num_processes`

  By passing a positive integer that indicates how many threads to use. It is `None` by default, and will use the result of `multiproccessing.cpu_count()`.

- `session`

  You can pass an alternative constructor or any callable that returns a `requests.Sesssion` like object. It will not be passed any arguments because a `requests.Session` does not accept any arguments.

class `requests_toolbelt.threaded.pool.`**`Pool`**(*job_queue*, *initializer=None*, *auth_generator=None*, *num_processes=None*, *session=<class 'requests.sessions.Session'>*)

Pool that manages the threads containing sessions.

Parameters

- **queue** (*queue.Queue*) – The queue you're expected to use to which you should add items.
- **initializer** (*collections.Callable*) – Function used to initialize an instance of `session`.
- **auth_generator** (*collections.Callable*) – Function used to generate new auth credentials for the session.
- **num_threads** (*int*) – Number of threads to create.
- **session** (*requests.Session*) –

`exceptions`()

Iterate over all the exceptions in the pool.

Returns Generator of `ThreadException`

classmethod `from_exceptions`(*exceptions*, ***kwargs*)

Create a `Pool` from an `ThreadExceptions`.

Provided an iterable that provides `ThreadException` objects, this classmethod will generate a new pool to retry the requests that caused the exceptions.

Parameters

- **exceptions** (*iterable*) – Iterable that returns `ThreadException`
- **kwargs** – Keyword arguments passed to the `Pool` initializer.

Returns An initialized `Pool` object.

Return type `Pool`

classmethod from_urls(*urls*, *request_kwargs=None*, ***kwargs*)
Create a `Pool` from an iterable of URLs.

Parameters

- **urls** (*iterable*) – Iterable that returns URLs with which we create a pool.
- **request_kwargs** (*dict*) – Dictionary of other keyword arguments to provide to the request method.
- **kwargs** – Keyword arguments passed to the `Pool` initializer.

Returns An initialized `Pool` object.

Return type `Pool`

get_exception()
Get an exception from the pool.

Return type `ThreadException`

get_response()
Get a response from the pool.

Return type `ThreadResponse`

join_all()
Join all the threads to the master thread.

responses()
Iterate over all the responses in the pool.

Returns Generator of `ThreadResponse`

class requests_toolbelt.threaded.pool.**ThreadResponse**(*request_kwargs*, *response*)
A wrapper around a requests Response object.

This will proxy most attribute access actions to the Response object. For example, if you wanted the parsed JSON from the response, you might do:

```
thread_response = pool.get_response()
json = thread_response.json()
```

class requests_toolbelt.threaded.pool.**ThreadException**(*request_kwargs*, *exception*)
A wrapper around an exception raised during a request.

This will proxy most attribute access actions to the exception object. For example, if you wanted the message from the exception, you might do:

```
thread_exc = pool.get_exception()
msg = thread_exc.message
```

2.7 Uploading Data

2.7.1 Streaming Multipart Data Encoder

Requests has support for multipart uploads, but the API means that using that functionality to build exactly the Multipart upload you want can be difficult or impossible. Additionally, when using Requests' Multipart upload functionality

all the data must be read into memory before being sent to the server. In extreme cases, this can make it impossible to send a file as part of a `multipart/form-data` upload.

The toolbelt contains a class that allows you to build multipart request bodies in exactly the format you need, and to avoid reading files into memory. An example of how to use it is like this:

```
import requests
from requests_toolbelt.multipart.encoder import MultipartEncoder

m = MultipartEncoder(
    fields={'field0': 'value', 'field1': 'value',
            'field2': ('filename', open('file.py', 'rb'), 'text/plain')}
    )

r = requests.post('http://httpbin.org/post', data=m,
                  headers={'Content-Type': m.content_type})
```

The `MultipartEncoder` has the `.to_string()` convenience method, as well. This method renders the multipart body into a string. This is useful when developing your code, allowing you to confirm that the multipart body has the form you expect before you send it on.

The toolbelt also provides a way to monitor your streaming uploads with the `MultipartEncoderMonitor`.

class `requests_toolbelt.multipart.encoder.`**`MultipartEncoder`**(*fields*, *boundary=None*, *encoding='utf-8'*)

The `MultipartEncoder` oject is a generic interface to the engine that will create a `multipart/form-data` body for you.

The basic usage is:

```
import requests
from requests_toolbelt import MultipartEncoder

encoder = MultipartEncoder({'field': 'value',
                            'other_field', 'other_value'})
r = requests.post('https://httpbin.org/post', data=encoder,
                  headers={'Content-Type': encoder.content_type})
```

If you do not need to take advantage of streaming the post body, you can also do:

```
r = requests.post('https://httpbin.org/post',
                  data=encoder.to_string(),
                  headers={'Content-Type': encoder.content_type})
```

If you want the encoder to use a specific order, you can use an OrderedDict or more simply, a list of tuples:

```
encoder = MultipartEncoder([('field', 'value'),
                            ('other_field', 'other_value')])
```

Changed in version 0.4.0.

You can also provide tuples as part values as you would provide them to requests' `files` parameter.

```
encoder = MultipartEncoder({
    'field': ('file_name', b'{"a": "b"}', 'application/json',
              {'X-My-Header': 'my-value'})
])
```

> **Warning:** This object will end up directly in `httplib`. Currently, `httplib` has a hard-coded read size of **8192 bytes**. This means that it will loop until the file has been read and your upload could take a while. This is **not** a bug in requests. A feature is being considered for this object to allow you, the user, to specify what size should be returned on a read. If you have opinions on this, please weigh in on this issue.

2.7.2 Monitoring Your Streaming Multipart Upload

If you need to stream your `multipart/form-data` upload then you're probably in the situation where it might take a while to upload the content. In these cases, it might make sense to be able to monitor the progress of the upload. For this reason, the toolbelt provides the `MultipartEncoderMonitor`. The monitor wraps an instance of a `MultipartEncoder` and is used exactly like the encoder. It provides a similar API with some additions:

- The monitor accepts a function as a callback. The function is called every time `requests` calls `read` on the monitor and passes in the monitor as an argument.
- The monitor tracks how many bytes have been read in the course of the upload.

You might use the monitor to create a progress bar for the upload. Here is an example using clint which displays the progress bar.

To use the monitor you would follow a pattern like this:

```
import requests
from requests_toolbelt.multipart import encoder

def my_callback(monitor):
    # Your callback function
    pass

e = encoder.MultipartEncoder(
    fields={'field0': 'value', 'field1': 'value',
            'field2': ('filename', open('file.py', 'rb'), 'text/plain')}
    )
m = encoder.MultipartEncoderMonitor(e, my_callback)

r = requests.post('http://httpbin.org/post', data=m,
                  headers={'Content-Type': m.content_type})
```

If you have a very simple use case you can also do:

```
import requests
from requests_toolbelt.multipart.encoder import MultipartEncoderMonitor

def my_callback(monitor):
    # Your callback function
    pass

m = MultipartEncoderMonitor.from_fields(
    fields={'field0': 'value', 'field1': 'value',
            'field2': ('filename', open('file.py', 'rb'), 'text/plain')},
    callback=my_callback
    )

r = requests.post('http://httpbin.org/post', data=m,
                  headers={'Content-Type': m.content_type})
```

class requests_toolbelt.multipart.encoder.**MultipartEncoderMonitor**(*encoder*, *callback=None*)

An object used to monitor the progress of a `MultipartEncoder`.

The `MultipartEncoder` should only be responsible for preparing and streaming the data. For anyone who wishes to monitor it, they shouldn't be using that instance to manage that as well. Using this class, they can monitor an encoder and register a callback. The callback receives the instance of the monitor.

To use this monitor, you construct your `MultipartEncoder` as you normally would.

```
from requests_toolbelt import (MultipartEncoder,
                               MultipartEncoderMonitor)
import requests

def callback(encoder, bytes_read):
    # Do something with this information
    pass

m = MultipartEncoder(fields={'field0': 'value0'})
monitor = MultipartEncoderMonitor(m, callback)
headers = {'Content-Type': montior.content_type}
r = requests.post('https://httpbin.org/post', data=monitor,
                  headers=headers)
```

Alternatively, if your use case is very simple, you can use the following pattern.

```
from requests_toolbelt import MultipartEncoderMonitor
import requests

def callback(encoder, bytes_read):
    # Do something with this information
    pass

monitor = MultipartEncoderMonitor.from_fields(
    fields={'field0': 'value0'}, callback
    )
headers = {'Content-Type': montior.content_type}
r = requests.post('https://httpbin.org/post', data=monitor,
                  headers=headers)
```

2.7.3 Streaming Data from a Generator

There are cases where you, the user, have a generator of some large quantity of data and you already know the size of that data. If you pass the generator to `requests` via the `data` parameter, `requests` will assume that you want to upload the data in chunks and set a `Transfer-Encoding` header value of `chunked`. Often times, this causes the server to behave poorly. If you want to avoid this, you can use the `StreamingIterator`. You pass it the size of the data and the generator.

```
import requests
from requests_toolbelt.streaming_iterator import StreamingIterator

generator = some_function()  # Create your generator
size = some_function_size()  # Get your generator's size
content_type = content_type()  # Get the content-type of the data

streamer = StreamingIterator(size, generator)
r = requests.post('https://httpbin.org/post', data=streamer,
                  headers={'Content-Type': content_type})
```

The streamer will handle your generator for you and buffer the data before passing it to `requests`.

Changed in version 0.4.0: File-like objects can be passed instead of a generator.

If, for example, you need to upload data being piped into standard in, you might otherwise do:

```
import requests
import sys

r = requests.post(url, data=sys.stdin)
```

This would stream the data but would use a chunked transfer-encoding. If instead, you know the length of the data that is being sent to `stdin` and you want to prevent the data from being uploaded in chunks, you can use the `StreamingIterator` to stream the contents of the file without relying on chunking.

```
import requests
from requests_toolbelt.streaming_iterator import StreamingIterator
import sys

stream = StreamingIterator(size, sys.stdin)
r = requests.post(url, data=stream,
                  headers={'Content-Type': content_type})
```

class requests_toolbelt.streaming_iterator.**StreamingIterator**(*size*, *iterator*, *encoding='utf-8'*)

This class provides a way of allowing iterators with a known size to be streamed instead of chunked.

In requests, if you pass in an iterator it assumes you want to use chunked transfer-encoding to upload the data, which not all servers support well. Additionally, you may want to set the content-length yourself to avoid this but that will not work. The only way to preempt requests using a chunked transfer-encoding and forcing it to stream the uploads is to mimic a very specific interace. Instead of having to know these details you can instead just use this class. You simply provide the size and iterator and pass the instance of StreamingIterator to requests via the data parameter like so:

```
from requests_toolbelt import StreamingIterator

import requests

# Let iterator be some generator that you already have and size be
# the size of the data produced by the iterator

r = requests.post(url, data=StreamingIterator(size, iterator))
```

You can also pass file-like objects to `StreamingIterator` in case requests can't determize the filesize itself. This is the case with streaming file objects like `stdin` or any sockets. Wrapping e.g. files that are on disk with `StreamingIterator` is unnecessary, because requests can determine the filesize itself.

Naturally, you should also set the *Content-Type* of your upload appropriately because the toolbelt will not attempt to guess that for you.

2.8 User-Agent Constructor

Having well-formed user-agent strings is important for the proper functioning of the web. Make server administators happy by generating yourself a nice user-agent string, just like Requests does! The output of the user-agent generator looks like this:

```
>>> import requests_toolbelt
>>> requests_toolbelt.user_agent('mypackage', '0.0.1')
'mypackage/0.0.1 CPython/2.7.5 Darwin/13.0.0'
```

The Python type and version, and the platform type and version, will accurately reflect the system that your program is running on. You can drop this easily into your program like this:

```
from requests_toolbelt import user_agent
from requests import Session

s = Session()
s.headers = {
    'User-Agent': user_agent('my_package', '0.0.1')
    }

r = s.get('https://api.github.com/users')
```

This will override the default Requests user-agent string for all of your HTTP requests, replacing it with your own.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

r