# requests_toolbelt Documentation

***Release 0.1.0***

**Ian Cordasco, Cory Benfield**

March 14, 2015

# Contents

**1 User Guide** **3**

1.1 Streaming Multipart Data Encoder . . . 3

1.2 User-Agent Constructor . . . 4

1.3 SSLAdapter . . . 5

1.4 GuessAuth . . . 5

**2 Indices and tables** **7**

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

# User Guide

The requests-toolbelt contains a number of unrelated tools and utilities for helping with use-cases that are not directly supported by the core requests library. This section of the documentation contains descriptions of the various utilities included in the toolbelt, and how to use them.

## 1.1 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:

```
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})
```

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.

### 1.1.1 Monitoring Your Streaming 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:

```
from requests_toolbelt import MultipartEncoder, MultipartEncoderMonitor
import requests

def my_callback(monitor):
    # Your callback function
    pass

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

```
from requests_toolbelt import MultipartEncoderMonitor
import requests

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})
```

## 1.2 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.

## 1.3 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 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.

## 1.4 GuessAuth

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

```
from requests_toolbelt import GuessAuth

import requests

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'))
```

This requires an additional request in case of basic auth, as usually basic auth is sent preemptively.

CHAPTER 2

# Indices and tables

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