Contents

1 Why? 3
2 Quick Start 5
  2.1 GET ................................................. 5
  2.2 POST .............................................. 5
3 Why not 100% requests-alike? 7
4 Feature Parity with Requests 9
5 Table of Contents 11
  5.1 Use Cases ........................................... 11
    5.1.1 Handling Streaming Responses .................. 11
    5.1.2 Query Parameters ............................... 11
    5.1.3 Auth ............................................ 12
    5.1.4 Redirects ...................................... 12
    5.1.5 Cookies ....................................... 13
    5.1.6 Agent Customization ......................... 14
  5.2 Testing Helpers .................................... 14
    5.2.1 Writing tests for HTTP clients ............... 14
    5.2.2 Writing tests for Twisted Web resources ....... 16
  5.3 API Reference ...................................... 16
    5.3.1 Making Requests ............................... 16
    5.3.2 Accessing Content ............................. 17
    5.3.3 HTTPClient Objects ......................... 18
    5.3.4 Augmented Response Objects ................ 18
    5.3.5 Test Helpers .................................. 19
    5.3.6 MultiPartProducer Objects ................. 21
6 Indices and tables .................................... 23
Python Module Index 25
treq depends on a recent Twisted and functions on Python 2.7 and Python 3.3+ (including PyPy).
requests by Kenneth Reitz is a wonderful library. I want the same ease of use when writing Twisted applications. treq is not of course a perfect clone of requests. I have tried to stay true to the do-what-I-mean spirit of the requests API and also kept the API familiar to users of Twisted and twisted.web.client.Agent on which treq is based.
CHAPTER 2

Quick Start

Installation:

```python
pip install treq
```

### 2.1 GET

```python
def main(reactor, *args):
    d = treq.get('http://httpbin.org/get')
    d.addCallback(print_response)
    return d
```

Full example: `basic_get.py`

### 2.2 POST

```python
def main(reactor, *args):
    d = treq.post('http://httpbin.org/post',
                  json.dumps({"msg": "Hello!"}).encode('ascii'),
                  headers={b'Content-Type': [b'application/json']})
    d.addCallback(print_response)
    return d
```

Full example: `basic_post.py`
Initially when I started off working on treq I thought the API should look exactly like requests except anything that would involve the network would return a Deferred.

Over time while attempting to mimic the requests API it became clear that not enough code could be shared between requests and treq for it to be worth the effort to translate many of the usage patterns from requests.

With the current version of treq I have tried to keep the API simple, yet remain familiar to users of Twisted and its lower-level HTTP libraries.
Feature Parity with Requests

Even though mimicking the requests API is not a goal, supporting most of its features is. Here is a list of requests features and their status in treq.

<table>
<thead>
<tr>
<th>Feature</th>
<th>requests</th>
<th>treq</th>
</tr>
</thead>
<tbody>
<tr>
<td>International Domains and URLs</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Keep-Alive &amp; Connection Pooling</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sessions with Cookie Persistence</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Browser-style SSL Verification</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Basic Authentication</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Digest Authentication</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Elegant Key/Value Cookies</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Automatic Decompression</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Unicode Response Bodies</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multipart File Uploads</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Connection Timeouts</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>.netrc support</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Python 2.6</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Python 2.7</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Python 3.x</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
5.1 Use Cases

5.1.1 Handling Streaming Responses

In addition to receiving responses with IResponse.deliverBody(), treq provides a helper function treq.collect() which takes a response and a single argument function which will be called with all new data available from the response. Much like IProtocol.dataReceived(), treq.collect() knows nothing about the framing of your data and will simply call your collector function with any data that is currently available.

Here is an example which simply a file object’s write method to treq.collect() to save the response body to a file.

```python
def download_file(reactor, url, destination_filename):
    destination = open(destination_filename, 'wb')
    d = treq.get(url)
    d.addCallback(treq.collect, destination.write)
    d.addBoth(lambda _: destination.close())
    return d
```

Full example: download_file.py

5.1.2 Query Parameters

treq.request() supports a params keyword argument which will be URL-encoded and added to the url argument in addition to any query parameters that may already exist.

The params argument may be either a dict or a list of (key, value) tuples.

If it is a dict then the values in the dict may either be a str value or a list of str values.
@inlineCallbacks
def main(reactor):
    print('List of tuples')
    resp = yield treq.get('http://httpbin.org/get',
        params=[('foo', 'bar'), ('baz', 'bax')])
    content = yield resp.text()
    print(content)

    print('Single value dictionary')
    resp = yield treq.get('http://httpbin.org/get',
        params={'foo': 'bar', 'baz': 'bax'})
    content = yield resp.text()
    print(content)

    print('Multi value dictionary')
    resp = yield treq.get('http://httpbin.org/get',
        params={'foo': ['bar', 'baz', 'bax']})
    content = yield resp.text()
    print(content)

    print('Mixed value dictionary')
    resp = yield treq.get('http://httpbin.org/get',
        params={'foo': ['bar', 'baz'], 'bax': 'quux'})
    content = yield resp.text()
    print(content)

    print('Preserved query parameters')
    resp = yield treq.get('http://httpbin.org/get?foo=bar',
        params={'baz': 'bax'})
    content = yield resp.text()
    print(content)

Full example: query_params.py

5.1.3 Auth

HTTP Basic authentication as specified in RFC 2617 is easily supported by passing an auth keyword argument to any of the request functions.

The auth argument should be a tuple of the form (‘username’, ‘password’).

def main(reactor, *args):
    d = treq.get('http://httpbin.org/basic-auth/treq/treq',
        auth=('treq', 'treq'))
    d.addCallback(print_response)
    return d

Full example: basic_auth.py

5.1.4 Redirects

treq handles redirects by default.
The following will print a 200 OK response.
def main(reactor, *args):
    d = treq.get('http://httpbin.org/redirect/1')
    d.addCallback(print_response)
    return d
react(main, [])

Full example: redirects.py

You can easily disable redirects by simply passing `allow_redirects=False` to any of the request methods.

def main(reactor, *args):
    d = treq.get('http://httpbin.org/redirect/1', allow_redirects=False)
    d.addCallback(print_response)
    return d
react(main, [])

Full example: disable_redirects.py

You can even access the complete history of treq response objects by calling the `history()` method on the response.

def main(reactor, *args):
    d = treq.get('http://httpbin.org/redirect/1')
    def cb(response):
        print('Response history: ')
        print(response.history())
        return print_response(response)
    d.addCallback(cb)
    d.addCallback(print_response)
    return d
Full example: response_history.py

5.1.5 Cookies

Cookies can be set by passing a `dict` or `cookielib.CookieJar` instance via the `cookies` keyword argument. Later cookies set by the server can be retrieved using the `cookies()` method.

The object returned by `cookies()` supports the same key/value access as `requests` cookies.

def main(reactor, *args):
    d = treq.get('http://httpbin.org/cookies/set?hello=world')
    def _get_jar(resp):
        jar = resp.cookies()
        print('The server set our hello cookie to: {}).format(jar['hello']))
        return treq.get('http://httpbin.org/cookies', cookies=jar)
    d.addCallback(_get_jar)
    d.addCallback(print_response)
    return d
Full example: using_cookies.py

5.1. Use Cases
5.1.6 Agent Customization

treq creates its own twisted.web.client.Agent with reasonable defaults, but you may want to provide your own custom agent. A custom agent can be passed to the various treq request methods using the agent keyword argument.

```python
custom_agent = Agent(reactor, connectTimeout=42)
treq.get(url, agent=custom_agent)
```

5.2 Testing Helpers

The treq.testing module provides some tools for testing both HTTP clients which use the treq API and implementations of the Twisted Web resource model.

5.2.1 Writing tests for HTTP clients

The StubTreq class implements the treq module interface (treq.get(), treq.post(), etc.) but runs all I/O via a MemoryReactor. It wraps a twisted.web.resource.IResource provider which handles each request.

You can wrap a pre-existing IResource provider, or write your own. For example, the twisted.web.resource.ErrorPage resource can produce an arbitrary HTTP status code. twisted.web.static.File can serve files or directories. And you can easily achieve custom responses by writing trivial resources yourself:

```python
@implementer(IResource)
class JsonResource(object):
    isLeaf = True  # NB: means getChildWithDefault will not be called

    def __init__(self, data):
        self.data = data

    def render(self, request):
        request.setHeader(b'Content-Type', b'application/json')
        return json.dumps(self.data).encode('utf-8')
```

However, those resources don’t assert anything about the request. The RequestSequence and StringStubbingResource classes make it easy to construct a resource which encodes the expected request and response pairs. Do note that most parameters to these functions must be bytes—it’s safest to use the b'' string syntax, which works on both Python 2 and 3.

For example:

```python
from twisted.internet import defer
from twisted.logger import Logger
from twisted.trial.unittest import SynchronousTestCase
from twisted.web import http
from treq.testing import StubTreq, HasHeaders
from treq.testing import RequestSequence, StringStubbingResource

log = Logger()

@defer.inlineCallbacks
def make_a_request(treq):
```

(continues on next page)
**5.2. Testing Helpers**

This may be run with `trial testing_seq.py`. Download: `testing_seq.py`.

Loosely matching the request

If you don’t care about certain parts of the request, you can pass `mock.ANY`, which compares equal to anything. This sequence matches a single GET request with any parameters or headers:
If you care about headers, use HasHeaders to make assertions about the headers present in the request. It compares equal to a superset of the headers specified, which helps make your test robust to changes in treq or Agent. Right now treq adds the Accept-Encoding: gzip header, but as support for additional compression methods is added, this may change.

5.2.2 Writing tests for Twisted Web resources

Since StubTreq wraps any resource, you can use it to test your server-side code as well. This is superior to calling your resource’s methods directly or passing mock objects, since it uses a real Agent to generate the request and a real Site to process the response. Thus, the request object your code interacts with is a real twisted.web.server.Request and behaves the same as it would in production.

Note that if your resource returns NOT_DONE_YET you must keep a reference to the RequestTraversalAgent and call its flush() method to spin the memory reactor once the server writes additional data before the client will receive it.

5.3 API Reference

This page lists all of the interfaces exposed by the treq package.

5.3.1 Making Requests

treq.request(method, url, **kwargs)

Make an HTTP request.

Parameters

- method (str) – HTTP method. Example: 'GET', 'HEAD', 'PUT', 'POST'.
- url (str) – http or https URL, which may include query arguments.
- headers (Headers or None) – Optional HTTP Headers to send with this request.
- params (dict w/ str or list/tuple of str values, list of 2-tuples, or None.) – Optional parameters to be append as the query string to the URL, any query string parameters in the URL already will be preserved.
- data (str, file-like, IBodyProducer, or None) – Optional request body.
- json (dict, list/tuple, int, string/unicode, bool, or None) – Optional JSON-serializable content to pass in body.
- reactor – Optional twisted reactor.
- persistent (bool) – Use persistent HTTP connections. Default: True
- allow_redirects (bool) – Follow HTTP redirects. Default: True
- auth (tuple of ('username', 'password').) – HTTP Basic Authentication information.
- cookies (dict or cookielib.CookieJar) – Cookies to send with this request. The HTTP kind, not the tasty kind.
• **timeout** *(int)* – Request timeout seconds. If a response is not received within this time-frame, a connection is aborted with `CancelledError`.

• **browser_like_redirects** *(bool)* – Use browser like redirects (i.e. Ignore RFC2616 section 10.3 and follow redirects from POST requests). Default: False

• **unbuffered** *(bool)* – Pass True to disable response buffering. By default treq buffers the entire response body in memory.

**Return type** Deferred that fires with an IResponse provider.

```python
treq.get(url, headers=None, **kwargs)
    Make a GET request.
    See treq.request()

treq.head(url, **kwargs)
    Make a HEAD request.
    See treq.request()

treq.post(url, data=None, **kwargs)
    Make a POST request.
    See treq.request()

treq.put(url, data=None, **kwargs)
    Make a PUT request.
    See treq.request()

treq.patch(url, data=None, **kwargs)
    Make a PATCH request.
    See treq.request()

treq.delete(url, **kwargs)
    Make a DELETE request.
    See treq.request()
```

### 5.3.2 Accessing Content

```python
treq.collect(response, collector)
    Incrementally collect the body of the response.

    This function may only be called once for a given response.

    **Parameters**
    
    • `response` *(IResponse)* – The HTTP response to collect the body from.
    
    • `collector` *(single argument callable)* – A callable to be called each time data is available from the response body.

    **Return type** Deferred that fires with None when the entire body has been read.
```

```python
treq.content(response)
    Read the contents of an HTTP response.

    This function may be called multiple times for a response, it uses a `WeakKeyDictionary` to cache the contents of the response.

    **Parameters** `response` *(IResponse)* – The HTTP Response to get the contents of.
Return type  Deferred that fires with the content as a str.

\texttt{treq.text\_content (response, encoding='ISO-8859-1')}  
Read the contents of an HTTP response and decode it with an appropriate charset, which may be guessed from the Content-Type header.

Parameters

- **response** (IResponse) – The HTTP Response to get the contents of.
- **encoding** (str) – A charset, such as UTF-8 or ISO-8859-1, used if the response does not specify an encoding.

Return type  Deferred that fires with a unicode string.

\texttt{treq.json\_content (response)}  
Read the contents of an HTTP response and attempt to decode it as JSON.

This function relies on \texttt{content()} and so may be called more than once for a given response.

Parameters **response** (IResponse) – The HTTP Response to get the contents of.

Return type  Deferred that fires with the decoded JSON.

### 5.3.3 HTTPClient Objects

The \texttt{treq.client.HTTPClient} class provides the same interface as the \texttt{treq} module itself.

\texttt{class treq.client.HTTPClient (agent, cookiejar=None, data\_to\_body\_producer=<InterfaceClass twisted.web.iweb.IBodyProducer>)}

- **delete** (url, **kwargs)
- **get** (url, **kwargs)
- **head** (url, **kwargs)
- **patch** (url, data=None, **kwargs)
- **post** (url, data=None, **kwargs)
- **put** (url, data=None, **kwargs)
- **request** (method, url, **kwargs)

### 5.3.4 Augmented Response Objects

\texttt{treq.request(), treq.get(), etc.} return an object which implements \texttt{twisted.web.iweb.IResponse}, plus a few additional convenience methods:

\texttt{class treq.response._Response}

- **collect** (collector)  
  Incrementally collect the body of the response, per \texttt{treq.collect()}.

  Parameters **collector** – A single argument callable that will be called with chunks of body data as it is received.

  Returns  A \texttt{Deferred} that fires when the entire body has been received.

- **content** ()  
  Read the entire body all at once, per \texttt{treq.content()}.  

---

Chapter 5. Table of Contents
Returns A `Deferred` that fires with a `bytes` object when the entire body has been received.

`json()` Collect the response body as JSON per `treq.json_content()`.

Return type Deferred that fires with the decoded JSON when the entire body has been read.

`text (encoding='ISO-8859-1')` Read the entire body all at once as text, per `treq.text_content()`.

Return type A `Deferred` that fires with a unicode string when the entire body has been received.

`history()` Get a list of all responses that (such as intermediate redirects), that ultimately ended in the current response. The responses are ordered chronologically.

Returns A list of `_Response` objects

`cookies()` Get a copy of this response’s cookies.

Return type `requests.cookies.RequestsCookieJar`

Inherited from `twisted.web.iweb.IResponse`:

Variables

- `version` –
- `code` –
- `phrase` –
- `headers` –
- `length` –
- `request` –
- `previousResponse` –

`deliverBody (protocol)`

`setPreviousResponse (response)`

### 5.3.5 Test Helpers

In-memory version of treq for testing.

**class treq.testing.HasHeaders (headers)**

Since Twisted adds headers to a request, such as the host and the content length, it’s necessary to test whether request headers CONTAIN the expected headers (the ones that are not automatically added by Twisted).

This wraps a set of headers, and can be used in an equality test against a superset if the provided headers. The headers keys are lowercased, and keys and values are compared in their bytes-encoded forms.

Headers should be provided as a mapping from strings or bytes to a list of strings or bytes.

**class treq.testing.RequestSequence (sequence, async_failure_reporter)**

For an example usage, see `RequestSequence.consume()`.

Takes a sequence of:

```
[[(method, url, params, headers, data), (code, headers, body)), ...]
```
Expects the requests to arrive in sequence order. If there are no more responses, or the request’s parameters do not match the next item’s expected request parameters, raises `AssertionError`.

For the expected request arguments:

- **method** should be `bytes` normalized to lowercase.
- **url** should be a `str` normalized as per the transformations in [URL normalization](https://en.wikipedia.org/wiki/URL_normalization) that (usually) preserve semantics. A URL to `http://something-that-looks-like-a-directory` would be normalized to `http://something-that-looks-like-a-directory/` and a URL to `http://something-that-looks-like-a-page/page.html` remains unchanged.
- **params** is a dictionary mapping `bytes` to `lists of bytes`
- **headers** is a dictionary mapping `bytes` to `lists of bytes` - note that `twisted.web.client.Agent` may add its own headers though, which are not guaranteed (for instance, `user-agent` or `content-length`), so it’s better to use some kind of matcher like `HasHeaders`.
- **data** is a `bytes`

For the response:

- **code** is an integer representing the HTTP status code to return
- **headers** is a dictionary mapping `bytes` to `bytes` or `lists of bytes`
- **body** is a `bytes`

**Variables**

- **sequence** (`list`) – The sequence of expected request arguments mapped to stubbed responses
- **async_failure_reporter** – A callable that takes a single message reporting failures—it’s asynchronous because it cannot just raise an exception—if it does, `Resource.render` will just convert that into a 500 response, and there will be no other failure reporting mechanism. Under Trial, this may be a `twisted.logger.Logger.error`, as Trial fails the test when an error is logged.

```python
consume(**kwds)
```

Usage:

```python
async_failures = []
sequence_stubs = RequestSequence([...], async_failures.append)
stub_treq = StubTreq(StringStubbingResource(sequence_stubs))
with sequence_stubs.consume(self.fail):  # self = unittest.TestCase
    stub_treq.get('http://fakeurl.com')
    stub_treq.get('http://another-fake-url.com')
self.assertEqual([], async_failures)
```

If there are still remaining expected requests to be made in the sequence, fails the provided test case.

**Parameters**

- **async_failure_reporter** – A callable that takes a single message reporting failures. This can just raise an exception - it does not need to be asynchronous, since the exception would not get raised within a Resource.

**Returns** a context manager that can be used to ensure all expected requests have been made.

```python
consumed()
```

**Returns** `bool` representing whether the entire sequence has been consumed. This is useful in tests to assert that the expected requests have all been made.
class treq.testing.RequestTraversalAgent (rootResource)
IAgent implementation that issues an in-memory request rather than going out to a real network socket.

flush()
Flush all data between pending client/server pairs.

This is only necessary if a Resource under test returns NOT_DONE_YET from its render method, making a response asynchronous. In that case, after each write from the server, pump() must be called so the client can see it.

request (method, uri, headers=None, bodyProducer=None)
Implement IAgent.request.

class treq.testing.StringStubbingResource (get_response_for)
A resource that takes a callable with 5 parameters (method, url, params, headers, data) and returns (code, headers, body).

The resource uses the callable to return a real response as a result of a request.

The parameters for the callable are:
- method, the HTTP method as bytes.
- url, the full URL of the request as text.
- params, a dictionary of query parameters mapping query keys lists of values (sorted alphabetically).
- headers, a dictionary of headers mapping header keys to a list of header values (sorted alphabetically).
- data, the request body as bytes.

The callable must return a tuple of (code, headers, body) where the code is the HTTP status code, the headers is a dictionary of bytes (unlike the headers parameter, which is a dictionary of lists), and body is a string that will be returned as the response body.

If there is a stubbing error, the return value is undefined (if an exception is raised, Resource will just eat it and return 500 in its place). The callable, or whomever creates the callable, should have a way to handle error reporting.

render (request)
Produce a response according to the stubs provided.

class treq.testing.StubTreq (resource)
A fake version of the treq module that can be used for testing that provides all the function calls exposed in treq.__all__.

Variables resource – A Resource object that provides the fake responses

5.3.6 MultiPartProducer Objects

treq.multipart.MultiPartProducer is used internally when making requests which involve files.

class treq.multipart.MultiPartProducer (fields, boundary=None, cooperator=<module 'twisted.internet.task' from '/home/docs/checkouts/readthedocs.org/user_builds/treq/envs/release-17.8.0/local/lib/python2.7/site-packages/twisted/internet/task.pyc'>)

MultiPartProducer takes parameters for a HTTP request and produces bytes in multipart/form-data format defined in RFC 2388 and RFC 2046.

The encoded request is produced incrementally and the bytes are written to a consumer.

Fields should have form: [[(parameter name, value), ...]
Accepted values:

- Unicode strings (in this case parameter will be encoded with utf-8)
- Tuples with (file name, content-type, IBodyProducer objects)

Since MultiPartProducer can accept objects like IBodyProducer which cannot be read from in an event-driven manner it uses uses a Cooperator instance to schedule reads from the underlying producers. Reading is also paused and resumed based on notifications from the IConsumer provider being written to.

Variables

- `_fields` – Sorted parameters, where all strings are enforced to be unicode and file objects stacked on bottom (to produce a human readable form-data request)
- `_cooperate` – A method like Cooperator.cooperate which is used to schedule all reads.
- `boundary` – The generated boundary used in form-data encoding

`pauseProducing()` ()
Temporarily suspend copying bytes from the input file to the consumer by pausing the CooperativeTask which drives that activity.

`resumeProducing()` ()
Undo the effects of a previous pauseProducing and resume copying bytes to the consumer by resuming the CooperativeTask which drives the write activity.

`startProducing` (consumer)
Start a cooperative task which will read bytes from the input file and write them to consumer. Return a Deferred which fires after all bytes have been written.

Parameters consumer – Any IConsumer provider

`stopProducing()` ()
Permanently stop writing bytes from the file to the consumer by stopping the underlying CooperativeTask.
CHAPTER 6

Indices and tables

- genindex
- modindex
- search
Python Module Index

t
  treq, 16
  treq.client, 18
  treq.multipart, 21
  treq.response, 18
  treq.testing, 19
Symbols

_Response (class in treq.response), 18

C
collect() (in module treq), 17
collect() (treq.response._Response method), 18
consume() (treq.testing.RequestSequence method), 20
consumed() (treq.testing.RequestSequence method), 20
content() (in module treq), 17
content() (treq.response._Response method), 18
cookies() (treq.response._Response method), 18

D
delete() (in module treq), 17
delete() (treq.client.HTTPClient method), 18
deliverBody() (treq.response._Response method), 19

F
flush() (treq.testing.RequestTraversalAgent method), 21

G
get() (in module treq), 17
get() (treq.client.HTTPClient method), 18

H
HasHeaders (class in treq.testing), 19
head() (in module treq), 17
head() (treq.client.HTTPClient method), 18
history() (treq.response._Response method), 19
HTTPClient (class in treq.client), 18

J
json() (treq.response._Response method), 19
json_content() (in module treq), 18

M
MultiPartProducer (class in treq.multipart), 21

P
patch() (in module treq), 17
patch() (treq.client.HTTPClient method), 18
pauseProducing() (treq.multipart.MultiPartProducer method), 22
post() (in module treq), 17
post() (treq.client.HTTPClient method), 18
put() (in module treq), 17
put() (treq.client.HTTPClient method), 18

R
render() (treq.testing.StringStubbingResource method), 21
request() (in module treq), 16
request() (treq.client.HTTPClient method), 18
request() (treq.testing.RequestTraversalAgent method), 21
RequestSequence (class in treq.testing), 19
RequestTraversalAgent (class in treq.testing), 20
resumeProducing() (treq.multipart.MultiPartProducer method), 22
RFC
RFC 2046, 21
RFC 2388, 21
RFC 2617, 12

S
setPreviousResponse() (treq.response._Response method), 19
startProducing() (treq.multipart.MultiPartProducer method), 22
stopProducing() (treq.multipart.MultiPartProducer method), 22
StringStubbingResource (class in treq.testing), 21
StubTreq (class in treq.testing), 21

T
text() (treq.response._Response method), 19
text_content() (in module treq), 18
treq (module), 16
rreq.client (module), 18
rreq.multipart (module), 21
rreq.response (module), 18
rreq.testing (module), 19