httsleep Documentation

Release 0.2.2

Aengus Walton

Jul 11, 2017

Contents

A python library for polling HTTP endpoints – batteries included!

httsleep aims to take care of any situation where you may need to poll a remote endpoint over HTTP, waiting for a certain response.

CHAPTER 1

Contents

Tutorial

Polling

httsleep polls a HTTP endpoint until it receives a response that matches a success condition. It returns a `requests.Response` object.

```
from httsleep import httsleep
response = httsleep('http://myendpoint/jobs/1', status_code=200)
```

In this example, httsleep will fire a HTTP GET request at `http://myendpoint/jobs/1` every 2 seconds, retrying a maximum of 50 times, until it gets a response with a status code of `200`.

We can change these defaults to poll once a minute, but a maximum of 10 times:

```
try:
    response = httsleep('http://myendpoint/jobs/1', status_code=200,
                        max_retries=10, polling_interval=60)
except StopIteration:
    print "Max retries has been exhausted!"
```

Similar to the Requests library, we can also set the `auth` to a `(username, password)` tuple and `headers` to a dict of headers if necessary. It is worth noting that these are provided as a convenience, since many APIs will require some form of authentication and client headers, and that httsleep doesn't duplicate the Requests library's API wholesale. Instead, you can pass a `requests.Request` object in place of the URL in more specific cases (e.g. polling using a POST request):

```
from requests import Request
req = Request('http://myendpoint/jobs/1', method='POST',
              data={'payload': 'here'})
response = httsleep(req, status_code=200)
```

If we're polling a server with a dodgy network connection, we might not want to break on a `requests.exceptions.ConnectionError`, but instead keep polling:

```
from requests.exceptions import ConnectionError
response = httsleep('http://myendpoint/jobs/1', status_code=200,
                    ignore_exceptions=[ConnectionError])
```

Conditions

Let's move on to specifying conditions. These are the conditions which, when met, cause httsleep to stop polling.

There are five conditions built in to httsleep:

- `status_code`
- `text`
- `json`
- `jsonpath`
- `callback`

The Basics

We've seen that `status_code` can be used to poll until a response with a certain status code is received. `text` and `json` are similar:

```
# Poll until the response body is the string "OK!":
httsleep('http://myendpoint/jobs/1', text="OK!")
# Poll until the json-decoded response has a certain value:
httsleep('http://myendpoint/jobs/1', json={'status': 'OK'})
```

If a `json` condition is specified but no JSON object could be decoded, a ValueError bubbles up. If needs be, this can be ignored by specifying `ignore_exceptions`.

JSONPath

The `json` condition is all well and good, but what if we're querying a resource on a RESTful API? The response may look something like the following:

```
{
    "id": 35872,
    "created": "2016-01-01 12:00:00",
    "updated": "2016-02-14 14:25:20",
    "status": "OK"
}
```

We won't necessarily know what the entire response (e.g. the object's ID, creation date, update date) will look like. This is where JSONPath comes into play. JSONPath makes it easy to focus on the information we want to compare in the JSON response and forget about everything else.

To assert that the `status` key of the JSON response is equal to `"OK"`, we can use the following JSONPath query:

```
httsleep('http://myendpoint/jobs/1',
         jsonpath=[{'expression': 'status', 'value': 'OK'}])
```

httsleep uses jsonpath-rw to evaluate JSONPath expressions. If you're familiar with this library, you can also use pre-compiled JSONPath expressions:

```
from jsonpath_rw.jsonpath import Fields
httsleep('http://myendpoint/jobs/1',
         jsonpath=[{'expression': Fields('status'), 'value': 'OK'}])
```

You might notice that the `jsonpath` kwarg value is a list. A response has only one status code, and only one body, but multiple JSONPath expressions might evaluate true for the JSON content returned. Therefore, you can string multiple JSONPaths together in a list. Logically, they will be evaluated with a boolean AND.

JSONPath is a highly powerful language, similar to XPath for XML. This section just skims the surface of what's possible with this language. To find out more about JSONPath and how to use it to build complex expressions, please refer to its documentation.

Callbacks

The last condition to have a look at is `callback`. This allows you to use your own function to evaluate the response and is intended for very specific cases where the other conditions might not be flexible enough.

A callback function should return `True` if the response matches. Any other return value will be interpreted as failure by httsleep, and it will keep polling.

Here is an example of a callback that makes sure the `last_scheduled_change` is in the past.

```
import datetime

def ensure_scheduled_change_in_past(response):
    data = response.json()
    last_scheduled_change = datetime.datetime.strptime(
        data['last_scheduled_change'], '%Y-%m-%d %H:%M:%S')
    if last_scheduled_change < datetime.datetime.utcnow():
        return True

httsleep('http://myendpoint/jobs/1', callback=ensure_scheduled_change_in_past)
```

Multiple Conditionals

It's possible to use multiple conditions simultaneously to assert many different things. Multiple conditions are joined using a boolean "AND".

For example, the following httsleep call will poll until a response with status code `200` AND an empty dict in the JSON body are received:

```
httsleep('http://myendpoint/jobs/1', status_code=200, json={})
```

The `until` kwarg

Until now, we've been specifying conditions by using direct kwargs. This can be a convenient shorthand for simple cases, but it's a little restrictive. It is also deprecated and will be removed in a future release.

There is another way: using the `until` kwarg. To demonstrate, *the previous example* could be rewritten as:

```
httsleep('http://myendpoint/jobs/1',
         until={'status_code': 200, 'json': {}})
```

One benefit of this is added readability – the client *sleeps until* a certain response is received. Another is the ability to chain conditions to form not just boolean ANDs, but also boolean ORs. More on that later in *Chaining Conditionals*.

Setting Alarms

Let's return to a previous example:

```
# Poll until the json-decoded response has a certain value:
httsleep('http://myendpoint/jobs/1', json={'status': 'OK'})
```

What if the job running on the remote server errors out and gets a status of `ERROR`? httsleep would keep polling the endpoint, waiting for a status of `OK`, until its `max_retries` had been exhausted – not exactly what we'd like to happen.

This is because no alarms have been set.

Alarms can be set using the `alarms` kwarg, just like success conditions can be set using the `until` kwarg. Every time it polls an endpoint, httsleep always checks whether any alarms are set, and if so, evaluates them. If the response matches an alarm condition, an *`httsleep.exceptions.Alarm`* exception is raised. If not, httsleep goes on and checks the success conditions.

Here is a version of the example above, modified so that it raises an *`httsleep.exceptions.Alarm`* if the job status is set to `ERROR`:

```
from httsleep.exceptions import Alarm
try:
    httsleep('http://myendpoint/jobs/1', json={'status': 'OK'},
             alarms={'json': {'status': 'ERROR'}})
except Alarm as e:
    print "Got a response with status ERROR!"
    print "Here's the response:", e.response
    print "And here's the alarm went off:", e.alarm
```

As can be seen here, the response object is stored in the exception, along with the alarm that was triggered.

Any conditions, or combination thereof, can be used to set alarms.

Chaining Conditionals and Alarms

We've seen that conditions can be joined together with a boolean "AND" by packing them into a single dictionary.

There are cases where we might want to join conditions using boolean "OR". In these cases, we simply use lists:

```
httsleep('http://myendpoint/jobs/1',
         until=[{'json': {'status': 'SUCCESS'}},
                {'json': {'status': 'PENDING'}}])
```

This means, "sleep until the json response is `{"status": "SUCCESS"}` OR `{"status": "PENDING"}`".

As always, we can use the same technique for alarms:

```
httsleep('http://myendpoint/jobs/1',
         until=[{'json': {'status': 'SUCCESS'}},
                {'json': {'status': 'PENDING'}}],
         alarms=[{'json': {'status': 'ERROR'}},
                 {'json': {'status': 'TIMEOUT'}}])
```

Putting it all together

As we've seen in this short tutorial, you can really squeeze a lot of flexibility out of *httsleep*.

We can see how far this can be taken in the next example:

```
until = {
    'status_code': 200,
    'jsonpath': [{'expression': 'status', 'value': 'OK'}]
}
alarms = [
    {'json': {'status': 'ERROR'}},
    {'jsonpath': [{'expression': 'status', 'value': 'UNKNOWN'},
                  {'expression': 'owner', 'value': 'Chris'}],
    'callback': is_job_really_failing},
    {'status_code': 404}
]
httsleep('http://myendpoint/jobs/1', until=until, alarms=alarms,
         max_retries=20)
```

Translated into English, this means:

- **Poll `http://myendpoint/jobs/1` – at most 20 times – until**
  - it returns a status code of `200`
  - AND the `status` key in its response has the value `OK`
- **but raise an error if**
  - the `status` key has the value `ERROR`
  - OR the `status` key has the value `UNKNOWN` AND the `owner` key has the value `Chris` AND the function `is_job_really_dying` returns `True`
  - OR the status code is 404

API Reference

`httsleep.`**`httsleep`**(*url_or_request*, *until=None*, *alarms=None*, *status_code=None*, *json=None*, *jsonpath=None*, *text=None*, *callback=None*, *auth=None*, *headers=None*, *polling_interval=2*, *max_retries=50*, *ignore_exceptions=None*, *loglevel=40*)

Convenience wrapper for the *`HttSleeper`* class. Creates a HttSleeper object and automatically runs it.

Returns `requests.Response` object.

class `httsleep.`**`HttSleeper`**(*url_or_request*, *until=None*, *alarms=None*, *status_code=None*, *json=None*, *jsonpath=None*, *text=None*, *callback=None*, *auth=None*, *headers=None*, *polling_interval=2*, *max_retries=50*, *ignore_exceptions=None*, *loglevel=40*)

Parameters

- **`url_or_request`** – either a string containing the URL to be polled, or a `requests.Request` object.
- **`until`** – a list of success conditions, respresented by dicts, or a single success condition dict.
- **`alarms`** – a list of error conditions, respresented by dicts, or a single error condition dict.

- **status_code** – deprecated. Shorthand for a success condition dependent on the response's status code.
- **json** – Deprecated. Shorthand for a success condition dependent on the response's JSON payload.
- **jsonpath** – Deprecated. Shorthand for a success condition dependent on the evaluation of a JSONPath expression.
- **text** – Deprecated. Shorthand for a success condition dependent on the response's body payload.
- **callback** – shorthand for a success condition dependent on a callback function that takes the response as an argument returning True.
- **auth** – a (username, password) tuple for HTTP authentication.
- **headers** – a dict of HTTP headers.
- **polling_interval** – how many seconds to sleep between requests.
- **max_retries** – the maximum number of retries to make, after which a StopIteration exception is raised.
- **ignore_exceptions** – a list of exceptions to ignore when polling the endpoint.
- **loglevel** – the loglevel to use. Defaults to *ERROR*.

`url_or_request` must be provided, along with at least one success condition (`until`).

run()

Polls the endpoint until either:

- a success condition in `self.until` is reached, in which case a `requests.Request` object is returned
- an error condition in `self.alarms` is encountered, in which case an `Alarm` exception is raised
- `self.max_retries` is reached, in which case a `StopIteration` exception is raised

Returns `requests.Response` object.

Exceptions

exception `httsleep.exceptions.`**Alarm**(*response*, *alarm_condition*)

Exception raised when an alarm condition has been met. Contains the following extra attributes:

- response: The response the matched the alarm condition
- alarm: The alarm condition that was triggered

CHAPTER 2

Motivation

Polling a remote endpoint over HTTP (e.g. waiting for a job to complete) is a very common task. The fact that there are no truly flexible polling libraries available leads to developers reproducing this boilerplate code time and time again.

A Simple Example

Maybe you want to just poll until you get a HTTP status code 200?

```
resp = httsleep('http://server/endpoint', status_code=200)
```

This example would be easily replaced with a few lines of Python code. However, most real-world cases aren't as simple as this, and your polling code ends up becoming more and more complicated – dealing with values in JSON payloads, cases where the remote server is unreachable, or cases where the job running remotely has errored out and we need to react accordingly.

httsleep aims to cover all of these cases – and more – by providing an array of validators (e.g. `status_code`, `json` and, most powerfully, `jsonpath`) which can be chained together logically, removing the burden of having to write any of this boilerplate code ever again.

A Real-World Example

"Poll my endpoint until it responds with the JSON payload `{'status': 'SUCCESS'}` and a HTTP status code 200, but raise an alarm if the HTTP status code is 500 or if the JSON payload is `{'status': 'TIMEOUT'}`. If a `ConnectionError` is thrown, ignore it, and give up after 20 attempts."

```
resp = httsleep('http://server/endpoint',
                until={'json': {'status': 'SUCCESS'},
                       'status_code': 200},
                alarms=[{'status_code': 500},
                        {'json': {'status': 'TIMEOUT'}}],
```

```
                ignore_exceptions=[ConnectionError],
                max_retries=20)
```

The Python code required to cover this logic would be significantly more complex, not to mention that it would require an extensive test suite be written.

This is the idea behind httsleep: outsource all of this logic to a library and not have to reimplement it for each different API you use.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index