Turq Documentation

Release 0.2.0

Vasiliy Faronov

Apr 04, 2017

Contents

Turq is a small HTTP server that is scriptable in a Python-based DSL. It is designed for mocking HTTP services quickly and interactively.

CHAPTER 1

Starting Turq

First you need to install it, normally from PyPI, for example:

```
$ pip install turq
```

Now you have a Python module called `turq`, and you can run it with:

```
$ python -m turq
```

This will start Turq on port 13085 by default, or you can choose another one with the `-p` option:

```
$ sudo python -m turq -p 80
```

Assuming your hostname is `machine.example`, and the port is 13085, you can now open the Turq console at `http://machine.example:13085/+turq/` or just `http://localhost:13085/+turq/`

The Turq console is where you post the rules that define your mock. Type in the code and hit "Commit", and Turq will start serving that.

CHAPTER 2

Rules structure

The code you post is pure Python code that is not sandboxed, which means you can import and use any modules if you wish so.

The code is interpreted right away. It should declare *rules* that will be applied to a matching request. Currently there is just one type of rule:

path (*path='*'*, *trailing_slash=True*)

This rule will be applied to every request whose path matches the `path` parameter, with an asterisk `*` meaning "zero or more of any characters". This is like routing in HTTP frameworks. The query string is not considered part of the path.

`path('/foo')` will match a request for `/foo/`, but `path('/foo', trailing_slash=False)` will not. A simple `path()` matches everything.

The *`path()`* function returns an object on which you then call methods to fill out your rule, i.e. how to respond to a matching request. Most methods can be daisy-chained. For example:

```
path('/index.html').html().gzip().expires('1 day')
```

You can also use a rule as a context manager:

```
with path('/index.html') as r:
    r.html()
    r.gzip()
    r.expires('1 day')
```

Matching rules are applied in the order they appear in your code. For example, given:

```
path().status(200)
path('/foo/bar/*').status(404)
path('/foo/*').status(500)
```

a request for `/foo/bar/baz` will result in status 500.

CHAPTER 3

Simple rule elements

`Rule`.**`status`**(*code*)
Set response status to *code*.

`Rule`.**`header`**(*name*, *value*, ***params*)
Set response header *name* to *value*.

If *params* are specified, they are appended as k=v pairs. For example:

```
header('Content-Type', 'text/html', charset='utf-8')
```

produces:

```
Content-Type: text/html; charset="utf-8"
```

`Rule`.**`add_header`**(*name*, *value*, ***params*)
Same as *`header()`*, but add the header, not replace it.

This can be used to send multiple headers with the same name, such as `Via` or `Set-Cookie` (but for the latter see *`cookie()`*).

`Rule`.**`body`**(*data*)
Set response entity body to *data*.

`Rule`.**`body_file`**(*path*)
Set response entity body to the contents of *path*.

The file at *path* is read when the rules are posted. *path* undergoes tilde expansion.

`Rule`.**`body_url`**(*url*)
Set response entity body to the contents of *url*.

The resource at *url* is fetched once and cached until Turq exits. Note that this method only sets the entity body. HTTP status and headers are not copied from *url*.

`Rule`.**`ctype`**(*value*)
Set response `Content-Type` to *value*.

`Rule.`**`text`**(*text='Hello world!'*)
Set up a `text/plain` response.

`Rule.`**`lots_of_text`**(*nbytes=20000*)
Set up a `text/plain` response with lots of text.

Lines of dummy text will be generated so that the entity body is very roughly *nbytes* in length.

`Rule.`**`html`**(*title='Hello world!'*, *text='This is Turq!'*)
Set up a `text/html` response.

A basic HTML page with *title* and a paragraph of *text* is served.

`Rule.`**`lots_of_html`**(*nbytes=20000*, *title='Hello world!'*)
Set up a `text/html` response with lots of text.

Like *`lots_of_text()`*, but wrapped in HTML paragraphs.

`Rule.`**`json`**(*data={'result': 'turq'}*, *jsonp=True*)
Set up a JSON or JSONP response.

data will be serialized into an `application/json` entity body. But if the request has a `callback` query parameter, *data* will be wrapped into a JSONP callback and served as `application/javascript`, unless you set *jsonp* to *False*.

`Rule.`**`js`**(*code='alert("Turq");'*)
Set up an `application/javascript` response.

`Rule.`**`xml`**(*code='<turq></turq>'*)
Set up an `application/xml` response.

`Rule.`**`redirect`**(*location*, *status=302*)
Set up a redirection response.

`Rule.`**`cookie`**(*name*, *value*, *max_age=None*, *expires=None*, *path=None*, *secure=False*, *http_only=False*)
Add a cookie *name* with *value*.

The other arguments correspond to parameters of the `Set-Cookie` header. If specified, *max_age* and *expires* should be strings. Nothing is escaped.

`Rule.`**`basic_auth`**(*realm='Turq'*)
Demand HTTP basic authentication (status code 401).

`Rule.`**`digest_auth`**(*realm='Turq'*, *nonce='twasbrillig'*)
Demand HTTP digest authentication (status code 401).

`Rule.`**`allow`**(**methods*)
Check the request method to be one of *methods* (case-insensitive).

If it isn't, send 405 Method Not Allowed with a plain-text error message.

`Rule.`**`cors`**()
Enable CORS on the response.

Currently this just sets `Access-Control-Allow-Origin: *`. Preflight requests are not yet handled.

`Rule.`**`expires`**(*when*)
Set the expiration time of the response.

when should be a specification of the number of minutes, hours or days, counted from the moment the rules are posted. Supported formats are: "10 min" or "10 minutes" or "5 h" or "5 hours" or "1 d" or "1 day".

`Rule.`**`gzip`**()
Apply `Content-Encoding: gzip` to the entity body.

`Accept-Encoding` of the request is ignored.

CHAPTER 4

Alternating responses

Within any rule, Turq can switch between multiple *sub-rules* for successive requests. A group of sub-rules is initiated with the `first()` call, which begins the sub-rule for the first request. You will also have zero or more `next()` calls and zero or one `then()` call. If you have `next()` but no `then()`, the cycle will eventually return to `first()` and start over.

Easier to explain by example. Here we alternate between "foo" and "bar":

```
with path('/') as r:
    r.first().text('foo')
    r.next().text('bar')
```

Here we arrive at "baz" and stay there:

```
with path('/') as r:
    r.first().text('foo')
    r.next().text('bar')
    r.then().text('baz')
```

Rule elements declared outside of `first()`, `next()` and `then()` will be applied to every response. For example, here, we send a custom `Server` header with every response:

```
with path('/') as r:
    r.header('Server', 'WonkyHTTPd/1.4.2b')
    r.first().text('foo')
    r.next().text('bar')
```

CHAPTER 5

Stochastic responses

Whereas `first()` et al. are deterministic, the *`maybe()`* call adds a stochastic dimension to the response.

`Rule.`**`maybe`**`(`*probability=0.1*`)`
: Add a sub-rule that will be applied with *probability*.

`Rule.`**`otherwise`**`()`
: Add a sub-rule that complements all *`maybe()`* rules.

  This is just a shortcut that adds a *maybe* sub-rule with a probability equal to 1 minus all currently defined *maybe*. Thus, it must come after all *maybe*.

This can be used to imitate occasional errors:

```
with path() as r:
    r.maybe().status(502).text('Bad Gateway')
    r.otherwise().html()
```

Probabilities don't have to cover everything:

```
with path() as r:
    r.html(text='Welcome to our site!')
    r.maybe(0.01).cookie('evilTracking', '12345')
```

CHAPTER 6

Parametrized responses

Sometimes you need the response to depend on the request. For example, suppose you have some crawler that fetches product info and expects the response to contain the requested product ID. You can do it easily with Turq:

```
path('/products').json(lambda req: {'id': req.query['id']})
```

Most rule elements that accept a simple value will also accept a function. The function is called with a *Request* object as the only argument.

class turq.**Request**(*method*, *path*, *query*, *headers*, *body*)

An HTTP request.

method

Request method.

path

The request path, excluding the query string.

query

A dictionary of parameters parsed from the query string. An empty dictionary if none can be parsed.

headers

An rfc822.Message-like mapping of request headers.

body

Request entity body as a *str* if there is one, otherwise *None*.

If you need even more logic, you can provide a custom handler function, attaching it to the rule by using the rule as a decorator:

```
@path('/products')
def process(req, r):
    if req.query['id'].startswith('SCR31-'):
        r.status(403).text('access to product info denied')
    else:
        r.json({'id': req.query['id']})
```

CHAPTER 7

Limitations

Turq does not provide full control over the HTTP exchange on the wire. For example:

- it always closes the connection after handling one request (and for this reason does not send `Content-Length` by default);
- the `Server` and `Date` response headers are always sent (but you can override them—in particular, set them to empty strings);
- you cannot change the HTTP version that is sent in the response status line.

If you need to tweak such things, you might be better off using the good old *netcat* or writing some custom code.

Index