Turq Documentation

Release 0.3.2.dev1

Vasiliy Faronov

Apr 05, 2017

Contents

1 User guide **3**
- 1.1 Quick start . . . 3
- 1.2 Programmatic use . . . 4
- 1.3 Using mitmproxy with Turq . . . 4
- 1.4 Known issues . . . 5

2 Examples **7**
- 2.1 Basics . . . 7
- 2.2 “RESTful” routing . . . 7
- 2.3 HTML pages . . . 7
- 2.4 Request details . . . 8
- 2.5 Response headers . . . 8
- 2.6 Custom status code and reason . . . 9
- 2.7 Redirection . . . 9
- 2.8 Authentication . . . 9
- 2.9 Body from file . . . 9
- 2.10 Inspecting requests . . . 9
- 2.11 Forwarding requests . . . 10
- 2.12 Cross-origin resource sharing . . . 10
- 2.13 Compression . . . 10
- 2.14 Random responses . . . 10
- 2.15 Response framing . . . 10
- 2.16 Streaming responses . . . 11
- 2.17 Handling `Expect: 100-continue` . . . 11
- 2.18 Custom methods . . . 11
- 2.19 Switching protocols . . . 12
- 2.20 Anything else . . . 12

3 History of changes **13**
- 3.1 0.3.1 - 2017-04-04 . . . 13
- 3.2 0.3.0 - 2017-04-04 . . . 13
- 3.3 0.2.0 - 2012-12-09 . . . 14
- 3.4 0.1.0 - 2012-11-17 . . . 14

Turq is a small HTTP server that can be scripted in a Python-based language. Use it to set up mock HTTP resources that respond with the status, headers, and body of your choosing. Turq is designed for quick interactive testing, but can be used in automated scenarios as well.

CHAPTER 1

User guide

Quick start

To run Turq, you need Python 3.4 or higher. Once you have that, install the turq package with pip:

```
$ pip3 install turq
```

Start Turq:

```
$ turq
```

You should see something like this:

```
18:22:19  turq  new rules installed
18:22:19  turq  mock on port 13085 - try http://pergamon:13085/
18:22:19  turq  editor on port 13086 - try http://pergamon:13086/
18:22:19  turq  editor password: QGOf9Y9Eqjvz4XhY4JA3U7hG (any username)
```

As you can see, Turq starts two HTTP servers. One is the *mock server* for the mocks you define. The other is the optional *rules editor* that makes writing mocks easier.

First you probably want to open the editor. By default, Turq listens on all network interfaces, so you can open the editor at `http://localhost:13086/` in your Web browser. Turq also tries to guess and print a URL that doesn't include `localhost`, which is useful when you run Turq on some remote machine via SSH.

Turq will ask you for the password that it generated and printed for you. You can leave the username field blank, it is ignored.

Warning: Anybody with access to the Turq editor can **execute arbitrary code** in the Turq process. The default password protection should keep you safe in most cases, but doesn't help against an active man-in-the-middle. If that's a problem, limit Turq to loopback with `--bind localhost`, or *run without the editor*.

In the editor, you define your mock by writing rules in the big code area, using the examples on the right as your guide. The default rules are just `error(404)`, which means that the mock server will respond with 404 (Not Found) to every request. Let's check that with curl:

```
$ curl -i http://pergamon:13085/some/page.html
HTTP/1.1 404 Not Found
content-type: text/plain; charset=utf-8
date: Tue, 04 Apr 2017 15:33:55 GMT
transfer-encoding: chunked

Error! Nothing matches the given URI
```

Keep an eye on the system console where you launched `turq` — all requests and responses are logged there:

```
19:01:30  turq.connection.1  new connection from 127.0.0.1
19:01:30  turq.request.1  > GET /some/page.html HTTP/1.1
19:01:30  turq.request.1  < HTTP/1.1 404 Not Found
```

When you are done, stop Turq by pressing Ctrl+C in the console.

That's it, basically. Check `turq --help` for command-line options, or read on for more hints on how to use Turq.

Programmatic use

Turq was designed for interactive use; it trades precision for convenience and simplicity. However, you can use it non-interactively if you like:

```
$ turq --no-editor --rules /path/to/rules.py
```

Give it a second to spin up, or just loop until you can `connect()` to it. Shut it down with SIGTERM like any other process:

```
$ pkill turq
```

It goes without saying that Turq can't be used anywhere near production.

Using mitmproxy with Turq

Put mitmproxy in front of Turq to:

- enable TLS (`https`) access to the mock server;
- inspect all requests and responses in detail;
- validate them with HTTPolice; and more.

Assuming Turq runs on the default port, use a command like this:

```
$ mitmproxy --port 13185 --reverse http://localhost:13085
```

Then tell your client to connect to port 13185 (`http` or `https`) instead of 13085.

Known issues

Password protection in the rules editor does not work well in some browsers. For example, you may randomly get "Connection error" in Internet Explorer. To avoid this, you can disable password protection with `-P ""`, but be sure to have some other protection instead.

The mock server doesn't send any cache-related headers by default. As a result, some browsers may cache your mocks, leading to strange results. You can disable caching in your rules:

```
add_header('Cache-Control', 'no-store')
```

Turq has limited options to control the addresses it listens on. You can forward its ports manually with socat or mitmproxy.

CHAPTER 2

Examples

The rules language of Turq is documented only by example.

Basics

```
# This is a comment. Normal Python syntax.
if path == '/hello':
    header('Content-Type', 'text/plain')
    body('Hello world!\r\n')
else:
    error(404)
```

"RESTful" routing

```
if route('/v1/products/:product_id'):
    if GET or HEAD:
        json({'id': int(product_id),
              'inStock': True})
    elif PUT:
        # Pretend that we saved it
        json(request.json)
    elif DELETE:
        status(204)    # No Content
```

HTML pages

To get a simple page:

```
html()
```

If you want to change the contents of the page, the full Dominate library is at your service (`dominate.tags` imported as `H`):

```
with html():
    H.h1('Welcome to our site')
    H.p('Have a look at our ',
        H.a('products', href='/products'))
```

To change the `<head>`:

```
with html() as document:
    with document.head:
        H.style('h1 {color: red}')
    H.h1('Welcome to our site')
```

Request details

```
if request.json:      # parsed JSON body
    name = request.json['name']
elif request.form:    # URL-encoded or multipart
    name = request.form['name']
elif query:       # query string parameters
    name = query['name']
else:
    raw_name = request.body      # raw bytes
    name = raw_name.decode('utf-8')

# Header names are case-insensitive
if 'json' in request.headers['Accept']:
    json({'hello': name})
else:
    text('Hello %s!\r\n' % name)
```

Response headers

`header()` *replaces* the given header, so this will send **only** `max-age`:

```
header('Cache-Control', 'public')
header('Cache-Control', 'max-age=3600')
```

To *add* a header instead:

```
add_header('Set-Cookie', 'sessionid=123456')
add_header('Set-Cookie', '__adtrack=abcdef')
```

Custom status code and reason

```
status(567, 'Server Fell Over')
text('Server crashed, sorry!\r\n')
```

Redirection

```
if path == '/':
    redirect('/index.html')
elif path == '/index.html':
    html()
```

`redirect()` sends 302 (Found) by default, but you can override:

```
redirect('/index.html', 307)
```

Authentication

To demand basic authentication:

```
basic_auth()
with html():
    H.h1('Super-secret page!')
```

This sends 401 (Unauthorized) unless the request had `Authorization` with the `Basic` scheme (credentials are ignored).

Similarly for digest:

```
digest_auth()
```

And for bearer:

```
bearer_auth()
```

Body from file

```
text(open('/etc/services'))
```

Inspecting requests

To see what the client sends, including headers (but not the raw body), put `debug()` somewhere early in your rules:

```
debug()
```

and watch the console output. Alternatively, for even more diagnostics, run Turq with the `--verbose` option.

Or use mitmproxy.

Forwarding requests

Turq can act as a gateway or "reverse proxy":

```
forward('httpbin.org', 80,  # host, port
        target)             # path + query string
# At this point, response from httpbin.org:80
# has been copied to Turq, and can be tweaked:
delete_header('Server')
add_header('Cache-Control', 'max-age=86400')
```

Turq uses TLS when connecting to port 443, but **ignores certificates**. You can override TLS like this:

```
forward('develop1.example', 8765,
        '/v1/articles', tls=True)
```

Cross-origin resource sharing

`cors()` adds the right `Access-Control-*` headers, and handles preflight requests automatically:

```
cors()
json({'some': 'data'})
```

For legacy systems, JSONP is also supported, reacting automatically to a `callback` query parameter:

```
json({'some': 'data'}, jsonp=True)
```

Compression

Call `gzip()` after setting the body:

```
with html():
    # 100 paragraphs of text
    for i in range(100):
        H.p(lorem_ipsum())
gzip()
```

Random responses

```
if maybe(0.1):    # 10% probability
    error(503)
else:
    html()
```

Response framing

By default, if the client supports it, Turq uses `Transfer-Encoding: chunked` and keeps the connection alive.

To use `Content-Length` instead of `Transfer-Encoding`, call `content_length()` after you've set the body:

```
text('Hello world!\r\n')
content_length()
```

To close the connection after sending the response:

```
add_header('Connection', 'close')
text('Hello world!\r\n')
```

Streaming responses

```
header('Content-Type', 'text/event-stream')
sleep(1)        # 1 second delay
chunk('data: my event 1\r\n\r\n')
sleep(1)
chunk('data: my event 2\r\n\r\n')
sleep(1)
chunk('data: my event 3\r\n\r\n')
```

Once you call `chunk()`, the response begins streaming. Any headers you set after that will be sent in the trailer part:

```
header('Content-Type', 'text/plain')
header('Trailer', 'Content-MD5')
chunk('Hello, ')
chunk('world!\n')
header('Content-MD5', '746308829575e17c3331bbcb00c0898b')
```

Handling `Expect: 100-continue`

```
with interim():
    status(100)

text('Resource updated OK')
```

In the above example, 100 (Continue) is sent immediately after the `interim()` block, but the final 200 (OK) response is sent only after reading the full request body.

If instead you want to send a response *before* reading the request body:

```
error(403)      # Forbidden
flush()
```

Custom methods

```
if method != 'FROBNICATE':
    error(405)    # Method Not Allowed
    header('Allow', 'FROBNICATE')
```

Switching protocols

```
if request.headers['Upgrade'] == 'QXTP':
    with interim():
        status(101)   # Switching Protocols
        header('Upgrade', 'QXTP')
        header('Connection', 'upgrade')
    send_raw('This is no longer HTTP!\r\n')
    send_raw('This is QXTP now!\r\n')
```

Anything else

In the end, Turq rules are just Python code that is not sandboxed, so you can import and use anything you like. For example, to send random binary data:

```
import os
header('Content-Type', 'application/octet-stream')
body(os.urandom(128))
```

CHAPTER 3

History of changes

0.3.1 - 2017-04-04

Packaging fixes.

0.3.0 - 2017-04-04

- Complete rewrite. Only the most notable changes are listed below.
- Requires Python 3.4 or higher.
- The rules language is completely different, simpler and more powerful.
- Notable new features of the rules language:
  - forwarding to other servers ("reverse proxy");
  - easy "RESTful" routing with path segments;
  - easy construction of arbitrary HTML pages (using Dominate);
  - CORS support now handles preflight requests automatically;
  - control over finer aspects of the protocol: streaming, 1xx responses, `Content-Length`, `Transfer-Encoding`, keep-alive.
- On the other hand, some features have been removed for now:
  - alternating responses (`first()`, `next()`, `then()`);
  - shortcuts for JavaScript and XML responses (`js()`, `xml()`).
- You can now choose which network interface Turq listens on, including IPv6.
- The Turq editor (formerly known as "console") now has automatic indentation and syntax highlighting.
- The Turq editor is now optional, listens on a separate port, and is protected with a password by default.

- Turq can now print more information to the (system) console, including request and response headers.
- Initial rules may now be read from a file at startup. This provides a simple way to use Turq programmatically.
- Turq can now handle multiple concurrent requests.

0.2.0 - 2012-12-09

- Stochastic responses (`maybe()`, `otherwise()`).
- Various features of the response can now be parametrized with lambdas.
- `body_file()` now expands tilde to the user's home directory.

0.1.0 - 2012-11-17

Initial release.