http.server — Base Classes for Implementing Web Servers

Purpose: http.server includes classes that can form the basis of a web server.

`http.server` uses classes from `socketserver` to create base classes for making HTTP servers. `HTTPServer` can be used directly, but the `BaseHTTPRequestHandler` is intended to be extended to handle each protocol method (GET, POST, etc.).

HTTP GET

To add support for an HTTP method in a request handler class, implement the method `do_METHOD()`, replacing `METHOD` with the name of the HTTP method (e.g., `do_GET()`, `do_POST()`, etc.). For consistency, the request handler methods take no arguments. All of the parameters for the request are parsed by `BaseHTTPRequestHandler` and stored as instance attributes of the request instance.

This example request handler illustrates how to return a response to the client, and some of the local attributes that can be useful in building the response.

```
# http_server_GET.py

from http.server import BaseHTTPRequestHandler
from urllib import parse


class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        parsed_path = parse.urlparse(self.path)
        message_parts = [
            'CLIENT VALUES:',
            'client_address={} ({})'.format(
                self.client_address,
                self.address_string()),
            'command={}'.format(self.command),
            'path={}'.format(self.path),
            'real path={}'.format(parsed_path.path),
            'query={}'.format(parsed_path.query),
            'request_version={}'.format(self.request_version),
            '',
            'SERVER VALUES:',
            'server_version={}'.format(self.server_version),
            'sys_version={}'.format(self.sys_version),
            'protocol_version={}'.format(self.protocol_version),
            '',
            'HEADERS RECEIVED:',
        ]
        for name, value in sorted(self.headers.items()):
            message_parts.append(
                '{}={}'.format(name, value.rstrip())
            )
        message_parts.append('')
        message = '\r\n'.join(message_parts)
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
        self.end_headers()
        self.wfile.write(message.encode('utf-8'))


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print('Starting server, use <Ctrl-C> to stop')
```

```
    server.serve_forever()
```

The message text is assembled and then written to `wfile`, the file handle wrapping the response socket. Each response needs a response code, set via `send_response()`. If an error code is used (404, 501, etc.), an appropriate default error message is included in the header, or a message can be passed with the error code.

To run the request handler in a server, pass it to the constructor of `HTTPServer`, as in the `__main__` processing portion of the sample script.

Then start the server:

```
$ python3 http_server_GET.py

Starting server, use <Ctrl-C> to stop
```

In a separate terminal, use `curl` to access it:

```
$ curl -v -i http://127.0.0.1:8080/?foo=bar

*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /?foo=bar HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.43.0
> Accept: */*
>
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:44:11 GMT

CLIENT VALUES:
client_address=('127.0.0.1', 52934) (127.0.0.1)
command=GET
path=/?foo=bar
real path=/
query=foo=bar
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept=*/*
Host=127.0.0.1:8080
User-Agent=curl/7.43.0
* Connection #0 to host 127.0.0.1 left intact
```

Note

The output produced by different versions of `curl` may vary. If running the examples produces different output, check the version number reported by `curl`.

HTTP POST

Supporting POST requests is a little more work, because the base class does not parse the form data automatically. The `cgi` module provides the `FieldStorage` class which knows how to parse the form, if it is given the correct inputs.

```
# http_server_POST.py

import cgi
from http.server import BaseHTTPRequestHandler
import io


class PostHandler(BaseHTTPRequestHandler):
```

```
    def do_POST(self):
        # Parse the form data posted
        form = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={
                'REQUEST_METHOD': 'POST',
                'CONTENT_TYPE': self.headers['Content-Type'],
            }
        )

        # Begin the response
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
        self.end_headers()

        out = io.TextIOWrapper(
            self.wfile,
            encoding='utf-8',
            line_buffering=False,
            write_through=True,
        )

        out.write('Client: {}\n'.format(self.client_address))
        out.write('User-agent: {}\n'.format(
            self.headers['user-agent']))
        out.write('Path: {}\n'.format(self.path))
        out.write('Form data:\n')

        # Echo back information about what was posted in the form
        for field in form.keys():
            field_item = form[field]
            if field_item.filename:
                # The field contains an uploaded file
                file_data = field_item.file.read()
                file_len = len(file_data)
                del file_data
                out.write(
                    '\tUploaded {} as {!r} ({} bytes)\n'.format(
                        field, field_item.filename, file_len)
                )
            else:
                # Regular form value
                out.write('\t{}={}\n'.format(
                    field, form[field].value))

        # Disconnect our encoding wrapper from the underlying
        # buffer so that deleting the wrapper doesn't close
        # the socket, which is still being used by the server.
        out.detach()


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), PostHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

Run the server in one window:

```
$ python3 http_server_POST.py

Starting server, use <Ctrl-C> to stop
```

The arguments to `curl` can include form data to be posted to the server by using the `-F` option. The last argument, `-F datafile=@http_server_GET.py`, posts the contents of the file `http_server_GET.py` to illustrate reading file data from the form.

```
$ curl -v http://127.0.0.1:8080/ -F name=dhellmann -F foo=bar \
-F datafile=@http_server_GET.py

*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> POST / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 1974
> Expect: 100-continue
> Content-Type: multipart/form-data;
boundary=------------------------a2b3c7485cf8def2
>
* Done waiting for 100-continue
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:53:48 GMT

Client: ('127.0.0.1', 53121)
User-agent: curl/7.43.0
Path: /
Form data:
    name=dhellmann
    Uploaded datafile as 'http_server_GET.py' (1612 bytes)
    foo=bar
* Connection #0 to host 127.0.0.1 left intact
```

Threading and Forking

HTTPServer is a simple subclass of socketserver.TCPServer, and does not use multiple threads or processes to handle requests. To add threading or forking, create a new class using the appropriate mix-in from socketserver.

```
# http_server_threads.py

from http.server import HTTPServer, BaseHTTPRequestHandler
from socketserver import ThreadingMixIn
import threading


class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
        self.end_headers()
        message = threading.currentThread().getName()
        self.wfile.write(message.encode('utf-8'))
        self.wfile.write(b'\n')


class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in a separate thread."""


if __name__ == '__main__':
    server = ThreadedHTTPServer(('localhost', 8080), Handler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

Run the server in the same way as the other examples.

```
$ python3 http_server_threads.py

Starting server, use <Ctrl-C> to stop
```

Each time the server receives a request, it starts a new thread or process to handle it:

```
$ curl http://127.0.0.1:8080/

Thread-1

$ curl http://127.0.0.1:8080/

Thread-2

$ curl http://127.0.0.1:8080/

Thread-3
```

Swapping `ForkingMixIn` for `ThreadingMixIn` would achieve similar results, using separate processes instead of threads.

Handling Errors

Handle errors by calling `send_error()`, passing the appropriate error code and an optional error message. The entire response (with headers, status code, and body) is generated automatically.

```
# http_server_errors.py

from http.server import BaseHTTPRequestHandler


class ErrorHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_error(404)


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), ErrorHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

In this case, a 404 error is always returned.

```
$ python3 http_server_errors.py

Starting server, use <Ctrl-C> to stop
```

The error message is reported to the client using an HTML document as well as the header to indicate an error code.

```
$ curl -i http://127.0.0.1:8080/

HTTP/1.0 404 Not Found
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:58:08 GMT
Connection: close
Content-Type: text/html;charset=utf-8
Content-Length: 447

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8">
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
        <p>Message: Not Found.</p>
        <p>Error code explanation: 404 - Nothing matches the
        given URI.</p>
    </body>
</html>
```

```
</html>
```

Setting Headers

The `send_header` method adds header data to the HTTP response. It takes two arguments: the name of the header and the value.

```
# http_server_send_header.py

from http.server import BaseHTTPRequestHandler
import time


class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.send_header(
            'Content-Type',
            'text/plain; charset=utf-8',
        )
        self.send_header(
            'Last-Modified',
            self.date_time_string(time.time())
        )
        self.end_headers()
        self.wfile.write('Response body\n'.encode('utf-8'))


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

This example sets the `Last-Modified` header to the current timestamp, formatted according to RFC 7231.

```
$ curl -i http://127.0.0.1:8080/

HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 21:00:54 GMT
Content-Type: text/plain; charset=utf-8
Last-Modified: Thu, 06 Oct 2016 21:00:54 GMT

Response body
```

The server logs the request to the terminal, like in the other examples.

```
$ python3 http_server_send_header.py

Starting server, use <Ctrl-C> to stop
127.0.0.1 - - [06/Oct/2016 17:00:54] "GET / HTTP/1.1" 200 -
```

Command Line Use

`http.server` includes a built-in server for serving files from the local file system. Start it from the command line using the `-m` option for the Python interpreter.

```
$ python3 -m http.server 8080

Serving HTTP on 0.0.0.0 port 8080 ...
127.0.0.1 - - [06/Oct/2016 17:12:48] "HEAD /index.rst HTTP/1.1" 200 -
```

The root directory of the server is the working directory where the server is started.

```
$ curl -I http://127.0.0.1:8080/index.rst
```

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 21:12:48 GMT
Content-type: application/octet-stream
Content-Length: 8285
Last-Modified: Thu, 06 Oct 2016 21:12:10 GMT
```

See also

- Standard library documentation for http.server
- `socketserver` – The `socketserver` module provides the base class that handles the raw socket connection.
- **RFC 7231** – "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content" includes a specification for the format of HTTP headers and dates.

base64 — Encode Binary Data with ASCII

http.cookies — HTTP Cookies

Quick Links

HTTP GET
HTTP POST
Threading and Forking
Handling Errors
Setting Headers
Command Line Use

This page was last updated 2016-12-31.

Navigation

base64 — Encode Binary Data with ASCII
http.cookies — HTTP Cookies

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example