Un peu plus loin avec `http.server`

Python: Simple HTTP Server on python.

https://islascruz.org/blog/2015/12/22/python-simple-http-server-on-python/

Posted on December 22, 2015 by marco

(Repost from old blog)

Python have several modules that help you to achieve your goals. This week, on my spare time that is getting every day more scarce I spend time figuring out how to create a Python Web Server, I was planing to use it over an application that I'm developing on ICT Consulting. At the end I didn't use it because I didn't want a "passive" comunication, but probably I will use this code on the CRM Desktop application that we use here.

Anyway, this code may be helpful for you too. I found that creating a small web server is really simple, It starts getting bigger as you add functions to that web server, but the basis is quite simple.

```
import os
import cgi
import sys
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

class customHTTPServer(BaseHTTPRequestHandler):
        def do_GET(self):
                self.send_response(200)
                self.send_header('Content-type', 'text/html')
                self.end_headers()
                self.wfile.write('<HTML><body>Get!</body></HTML>')
                return

        def do_POST(self):
                global rootnode
                ctype,pdict = cgi.parse_header(self.headers.getheader('Content-
type'))
                if ctype == 'multipart/form-data':
                        query = cgi.parse_multipart(self.rfile, pdict)
                self.send_response(301)
                self.end_headers()
                self.wfile.write('Post!')


def main():
        try:
                server = HTTPServer(('',8080),customHTTPServer)
                print 'server started at port 8080'
                server.serve_forever()
        except KeyboardInterrupt:
                server.socket.close()

if __name__=='__main__':
        main()
```

There are two main methods in our small server: do_GET and do_POST, you can figure out what this methods do. Get is quite simple, Post is used to send data to the server, as an example, the file uploading. This is done via POST and many times using “multipat/form-data” as the content type. The Post method here handles that.

Now that you have a custom server, how can you check it?, well, to check GET you can call it from the web browser. For the POST stuff, you can create a simple web form and using your web browser as “action” on it. However, this code can help you:

There are two main methods in our small server: do_GET and do_POST, you can figure out what this methods do. Get is quite simple, Post is used to send data to the server, as an example, the file uploading. This is done via POST and many times using “multipat/form-data” as the content type. The Post method here handles that.

Now that you have a custom server, how can you check it?, well, to check GET you can call it from the web browser. For the POST stuff, you can create a simple web form and using your web browser as “action” on it. However, this code can help you:

```
import urllib2
import urllib
import time
import httplib, mimetypes

HOST = '127.0.0.1'
PORT = '8080'

def post_multipart(host, port, selector, fields, files):
        """
        Post fields and files to an http host as multipart/form-data.
        fields is a sequence of (name, value) elements for regular form fields.
        files is a sequence of (name, filename, value) elements for data to be
uploaded as files
        Return the server's response page.
        """
        content_type, body = encode_multipart_formdata(fields, files)
        h = httplib.HTTP(host, port)
        h.putrequest('POST', '/cgi-bin/query')
        h.putheader('content-type', content_type)
        h.putheader('content-length', str(len(body)))
        h.endheaders()
        h.send(body)
        #errcode, errmsg, headers = h.getreply()
        h.getreply()
        return h.file.read()

def encode_multipart_formdata(fields, files):
        """
        fields is a sequence of (name, value) elements for regular form fields.
        files is a sequence of (name, filename, value) elements for data to be
uploaded as files
        Return (content_type, body) ready for httplib.HTTP instance
        """
        BOUNDARY = '----------ThIs_Is_tHe_bouNdaRY_$'
        CRLF = '\r\n'
        L = []
        if fields:
                for (key, value) in fields:
```

```
                        L.append('--' + BOUNDARY)
                        L.append('Content-Disposition: form-data; name="%s"' %
key)
                        L.append('')
                        L.append(value)
        if files:
                for (key, filename, value) in files:
                        L.append('--' + BOUNDARY)
                        L.append('Content-Disposition: form-data; name="%s";
filename="%s"' % (key, filename))
                        L.append('Content-Type: %s' %
get_content_type(filename))
                        L.append('')
                        L.append(value)
        L.append('--' + BOUNDARY + '--')
        L.append('')
        body = CRLF.join(L)
        content_type = 'multipart/form-data; boundary=%s' % BOUNDARY
        return content_type, body

def get_content_type(filename):
        return mimetypes.guess_type(filename)[0] or 'application/octet-stream'

def test():
        print  post_multipart(HOST, PORT, 'markuz',
                                        ( ('username','markuz'),
('another_field','another value')),
                                        (('query','query','Query'), ),
                                    )

if __name__ == '__main__':
     test()
```

UniIsland/SimpleHTTPServerWithUpload.py

https://gist.github.com/UniIsland/3346170

```
#!/usr/bin/env python

"""Simple HTTP Server With Upload.
This module builds on BaseHTTPServer by implementing the standard GET
and HEAD requests in a fairly straightforward manner.
"""


__version__ = "0.1"
__all__ = ["SimpleHTTPRequestHandler"]
__author__ = "bones7456"
__home_page__ = "http://li2z.cn/"

import os
import posixpath
import BaseHTTPServer
import urllib
import cgi
import shutil
import mimetypes
import re
try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO


class SimpleHTTPRequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):

    """Simple HTTP request handler with GET/HEAD/POST commands.
    This serves files from the current directory and any of its
    subdirectories.  The MIME type for files is determined by
    calling the .guess_type() method. And can reveive file uploaded
    by client.
    The GET/HEAD/POST requests are identical except that the HEAD
    request omits the actual contents of the file.
    """

    server_version = "SimpleHTTPWithUpload/" + __version__

    def do_GET(self):
        """Serve a GET request."""
        f = self.send_head()
        if f:
            self.copyfile(f, self.wfile)
            f.close()

    def do_HEAD(self):
        """Serve a HEAD request."""
        f = self.send_head()
        if f:
            f.close()
```

```
    def do_POST(self):
        """Serve a POST request."""
        r, info = self.deal_post_data()
        print r, info, "by: ", self.client_address
        f = StringIO()
        f.write('<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">')
        f.write("<html>\n<title>Upload Result Page</title>\n")
        f.write("<body>\n<h2>Upload Result Page</h2>\n")
        f.write("<hr>\n")
        if r:
            f.write("<strong>Success:</strong>")
        else:
            f.write("<strong>Failed:</strong>")
        f.write(info)
        f.write("<br><a href=\"%s\">back</a>" % self.headers['referer'])
        f.write("<hr><small>Powerd By: bones7456, check new version at ")
        f.write("<a href=\"http://li2z.cn/?s=SimpleHTTPServerWithUpload\">")
        f.write("here</a>.</small></body>\n</html>\n")
        length = f.tell()
        f.seek(0)
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(length))
        self.end_headers()
        if f:
            self.copyfile(f, self.wfile)
            f.close()

    def deal_post_data(self):
        boundary = self.headers.plisttext.split("=")[1]
        remainbytes = int(self.headers['content-length'])
        line = self.rfile.readline()
        remainbytes -= len(line)
        if not boundary in line:
            return (False, "Content NOT begin with boundary")
        line = self.rfile.readline()
        remainbytes -= len(line)
        fn = re.findall(r'Content-Disposition.*name="file"; filename="(.*)"',
line)
        if not fn:
            return (False, "Can't find out file name...")
        path = self.translate_path(self.path)
        fn = os.path.join(path, fn[0])
        line = self.rfile.readline()
        remainbytes -= len(line)
        line = self.rfile.readline()
        remainbytes -= len(line)
        try:
            out = open(fn, 'wb')
        except IOError:
            return (False, "Can't create file to write, do you have permission
to write?")

        preline = self.rfile.readline()
        remainbytes -= len(preline)
        while remainbytes > 0:
            line = self.rfile.readline()
            remainbytes -= len(line)
            if boundary in line:
                preline = preline[0:-1]
```

```
                if preline.endswith('\r'):
                    preline = preline[0:-1]
                out.write(preline)
                out.close()
                return (True, "File '%s' upload success!" % fn)
            else:
                out.write(preline)
                preline = line
        return (False, "Unexpect Ends of data.")

    def send_head(self):
        """Common code for GET and HEAD commands.
        This sends the response code and MIME headers.
        Return value is either a file object (which has to be copied
        to the outputfile by the caller unless the command was HEAD,
        and must be closed by the caller under all circumstances), or
        None, in which case the caller has nothing further to do.
        """
        path = self.translate_path(self.path)
        f = None
        if os.path.isdir(path):
            if not self.path.endswith('/'):
                # redirect browser - doing basically what apache does
                self.send_response(301)
                self.send_header("Location", self.path + "/")
                self.end_headers()
                return None
            for index in "index.html", "index.htm":
                index = os.path.join(path, index)
                if os.path.exists(index):
                    path = index
                    break
            else:
                return self.list_directory(path)
        ctype = self.guess_type(path)
        try:
            # Always read in binary mode. Opening files in text mode may cause
            # newline translations, making the actual size of the content
            # transmitted *less* than the content-length!
            f = open(path, 'rb')
        except IOError:
            self.send_error(404, "File not found")
            return None
        self.send_response(200)
        self.send_header("Content-type", ctype)
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified", self.date_time_string(fs.st_mtime))
        self.end_headers()
        return f

    def list_directory(self, path):
        """Helper to produce a directory listing (absent index.html).
        Return value is either a file object, or None (indicating an
        error).  In either case, the headers are sent, making the
        interface the same as for send_head().
        """
        try:
            list = os.listdir(path)
        except os.error:
            self.send_error(404, "No permission to list directory")
```

```
            return None
        list.sort(key=lambda a: a.lower())
        f = StringIO()
        displaypath = cgi.escape(urllib.unquote(self.path))
        f.write('<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">')
        f.write("<html>\n<title>Directory listing for %s</title>\n" %
displaypath)
        f.write("<body>\n<h2>Directory listing for %s</h2>\n" % displaypath)
        f.write("<hr>\n")
        f.write("<form ENCTYPE=\"multipart/form-data\" method=\"post\">")
        f.write("<input name=\"file\" type=\"file\"/>")
        f.write("<input type=\"submit\" value=\"upload\"/></form>\n")
        f.write("<hr>\n<ul>\n")
        for name in list:
            fullname = os.path.join(path, name)
            displayname = linkname = name
            # Append / for directories or @ for symbolic links
            if os.path.isdir(fullname):
                displayname = name + "/"
                linkname = name + "/"
            if os.path.islink(fullname):
                displayname = name + "@"
                # Note: a link to a directory displays with @ and links with /
            f.write('<li><a href="%s">%s</a>\n'
                    % (urllib.quote(linkname), cgi.escape(displayname)))
        f.write("</ul>\n<hr>\n</body>\n</html>\n")
        length = f.tell()
        f.seek(0)
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(length))
        self.end_headers()
        return f

    def translate_path(self, path):
        """Translate a /-separated PATH to the local filename syntax.
        Components that mean special things to the local file system
        (e.g. drive or directory names) are ignored.  (XXX They should
        probably be diagnosed.)
        """
        # abandon query parameters
        path = path.split('?',1)[0]
        path = path.split('#',1)[0]
        path = posixpath.normpath(urllib.unquote(path))
        words = path.split('/')
        words = filter(None, words)
        path = os.getcwd()
        for word in words:
            drive, word = os.path.splitdrive(word)
            head, word = os.path.split(word)
            if word in (os.curdir, os.pardir): continue
            path = os.path.join(path, word)
        return path

    def copyfile(self, source, outputfile):
        """Copy all data between two file objects.
        The SOURCE argument is a file object open for reading
        (or anything with a read() method) and the DESTINATION
        argument is a file object open for writing (or
        anything with a write() method).
        The only reason for overriding this would be to change
```

```
        the block size or perhaps to replace newlines by CRLF
        -- note however that this the default server uses this
        to copy binary data as well.
        """
        shutil.copyfileobj(source, outputfile)

    def guess_type(self, path):
        """Guess the type of a file.
        Argument is a PATH (a filename).
        Return value is a string of the form type/subtype,
        usable for a MIME Content-type header.
        The default implementation looks the file's extension
        up in the table self.extensions_map, using application/octet-stream
        as a default; however it would be permissible (if
        slow) to look inside the data to make a better guess.
        """

        base, ext = posixpath.splitext(path)
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        ext = ext.lower()
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        else:
            return self.extensions_map['']

    if not mimetypes.inited:
        mimetypes.init() # try to read system mime.types
    extensions_map = mimetypes.types_map.copy()
    extensions_map.update({
        '': 'application/octet-stream', # Default
        '.py': 'text/plain',
        '.c': 'text/plain',
        '.h': 'text/plain',
        })


def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)

if __name__ == '__main__':
test()
```

touilleMan/SimpleHTTPServerWithUpload.py

forked from UniIsland/SimpleHTTPServerWithUpload.py

Last active 19 days ago

```
#!/usr/bin/env python3

"""Simple HTTP Server With Upload.
This module builds on BaseHTTPServer by implementing the standard GET
and HEAD requests in a fairly straightforward manner.
see: https://gist.github.com/UniIsland/3346170
"""


__version__ = "0.1"
__all__ = ["SimpleHTTPRequestHandler"]
__author__ = "bones7456"
__home_page__ = "http://li2z.cn/"

import os
import posixpath
import http.server
import urllib.request, urllib.parse, urllib.error
import cgi
import shutil
import mimetypes
import re
from io import BytesIO


class SimpleHTTPRequestHandler(http.server.BaseHTTPRequestHandler):

    """Simple HTTP request handler with GET/HEAD/POST commands.
    This serves files from the current directory and any of its
    subdirectories.  The MIME type for files is determined by
    calling the .guess_type() method. And can reveive file uploaded
    by client.
    The GET/HEAD/POST requests are identical except that the HEAD
    request omits the actual contents of the file.
    """

    server_version = "SimpleHTTPWithUpload/" + __version__

    def do_GET(self):
        """Serve a GET request."""
        f = self.send_head()
        if f:
            self.copyfile(f, self.wfile)
            f.close()

    def do_HEAD(self):
        """Serve a HEAD request."""
        f = self.send_head()
        if f:
            f.close()

    def do_POST(self):
```

```
        """Serve a POST request."""
        r, info = self.deal_post_data()
        print((r, info, "by: ", self.client_address))
        f = BytesIO()
        f.write(b'<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">')
        f.write(b"<html>\n<title>Upload Result Page</title>\n")
        f.write(b"<body>\n<h2>Upload Result Page</h2>\n")
        f.write(b"<hr>\n")
        if r:
            f.write(b"<strong>Success:</strong>")
        else:
            f.write(b"<strong>Failed:</strong>")
        f.write(info.encode())
        f.write(("<br><a href=\"%s\">back</a>" %
self.headers['referer']).encode())
        f.write(b"<hr><small>Powerd By: bones7456, check new version at ")
        f.write(b"<a href=\"http://li2z.cn/?s=SimpleHTTPServerWithUpload\">")
        f.write(b"here</a>.</small></body>\n</html>\n")
        length = f.tell()
        f.seek(0)
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(length))
        self.end_headers()
        if f:
            self.copyfile(f, self.wfile)
            f.close()

    def deal_post_data(self):
        content_type = self.headers['content-type']
        if not content_type:
            return (False, "Content-Type header doesn't contain boundary")
        boundary = content_type.split("=")[1].encode()
        remainbytes = int(self.headers['content-length'])
        line = self.rfile.readline()
        remainbytes -= len(line)
        if not boundary in line:
            return (False, "Content NOT begin with boundary")
        line = self.rfile.readline()
        remainbytes -= len(line)
        fn = re.findall(r'Content-Disposition.*name="file"; filename="(.*)"',
line.decode())
        if not fn:
            return (False, "Can't find out file name...")
        path = self.translate_path(self.path)
        fn = os.path.join(path, fn[0])
        line = self.rfile.readline()
        remainbytes -= len(line)
        line = self.rfile.readline()
        remainbytes -= len(line)
        try:
            out = open(fn, 'wb')
        except IOError:
            return (False, "Can't create file to write, do you have permission
to write?")

        preline = self.rfile.readline()
        remainbytes -= len(preline)
        while remainbytes > 0:
            line = self.rfile.readline()
            remainbytes -= len(line)
```

```
            if boundary in line:
                preline = preline[0:-1]
                if preline.endswith(b'\r'):
                    preline = preline[0:-1]
                out.write(preline)
                out.close()
                return (True, "File '%s' upload success!" % fn)
            else:
                out.write(preline)
                preline = line
        return (False, "Unexpect Ends of data.")

    def send_head(self):
        """Common code for GET and HEAD commands.
        This sends the response code and MIME headers.
        Return value is either a file object (which has to be copied
        to the outputfile by the caller unless the command was HEAD,
        and must be closed by the caller under all circumstances), or
        None, in which case the caller has nothing further to do.
        """
        path = self.translate_path(self.path)
        f = None
        if os.path.isdir(path):
            if not self.path.endswith('/'):
                # redirect browser - doing basically what apache does
                self.send_response(301)
                self.send_header("Location", self.path + "/")
                self.end_headers()
                return None
            for index in "index.html", "index.htm":
                index = os.path.join(path, index)
                if os.path.exists(index):
                    path = index
                    break
            else:
                return self.list_directory(path)
        ctype = self.guess_type(path)
        try:
            # Always read in binary mode. Opening files in text mode may cause
            # newline translations, making the actual size of the content
            # transmitted *less* than the content-length!
            f = open(path, 'rb')
        except IOError:
            self.send_error(404, "File not found")
            return None
        self.send_response(200)
        self.send_header("Content-type", ctype)
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified", self.date_time_string(fs.st_mtime))
        self.end_headers()
        return f

    def list_directory(self, path):
        """Helper to produce a directory listing (absent index.html).
        Return value is either a file object, or None (indicating an
        error).  In either case, the headers are sent, making the
        interface the same as for send_head().
        """
        try:
            list = os.listdir(path)
```

```
        except os.error:
            self.send_error(404, "No permission to list directory")
            return None
        list.sort(key=lambda a: a.lower())
        f = BytesIO()
        displaypath = cgi.escape(urllib.parse.unquote(self.path))
        f.write(b'<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">')
        f.write(("<html>\n<title>Directory listing for %s</title>\n" %
displaypath).encode())
        f.write(("<body>\n<h2>Directory listing for %s</h2>\n" %
displaypath).encode())
        f.write(b"<hr>\n")
        f.write(b"<form ENCTYPE=\"multipart/form-data\" method=\"post\">")
        f.write(b"<input name=\"file\" type=\"file\"/>")
        f.write(b"<input type=\"submit\" value=\"upload\"/></form>\n")
        f.write(b"<hr>\n<ul>\n")
        for name in list:
            fullname = os.path.join(path, name)
            displayname = linkname = name
            # Append / for directories or @ for symbolic links
            if os.path.isdir(fullname):
                displayname = name + "/"
                linkname = name + "/"
            if os.path.islink(fullname):
                displayname = name + "@"
                # Note: a link to a directory displays with @ and links with /
            f.write(('<li><a href="%s">%s</a>\n'
                    % (urllib.parse.quote(linkname),
cgi.escape(displayname))).encode())
        f.write(b"</ul>\n<hr>\n</body>\n</html>\n")
        length = f.tell()
        f.seek(0)
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(length))
        self.end_headers()
        return f

    def translate_path(self, path):
        """Translate a /-separated PATH to the local filename syntax.
        Components that mean special things to the local file system
        (e.g. drive or directory names) are ignored.  (XXX They should
        probably be diagnosed.)
        """
        # abandon query parameters
        path = path.split('?',1)[0]
        path = path.split('#',1)[0]
        path = posixpath.normpath(urllib.parse.unquote(path))
        words = path.split('/')
        words = [_f for _f in words if _f]
        path = os.getcwd()
        for word in words:
            drive, word = os.path.splitdrive(word)
            head, word = os.path.split(word)
            if word in (os.curdir, os.pardir): continue
            path = os.path.join(path, word)
        return path

    def copyfile(self, source, outputfile):
        """Copy all data between two file objects.
        The SOURCE argument is a file object open for reading
```

```
        (or anything with a read() method) and the DESTINATION
        argument is a file object open for writing (or
        anything with a write() method).
        The only reason for overriding this would be to change
        the block size or perhaps to replace newlines by CRLF
        -- note however that this the default server uses this
        to copy binary data as well.
        """
        shutil.copyfileobj(source, outputfile)

    def guess_type(self, path):
        """Guess the type of a file.
        Argument is a PATH (a filename).
        Return value is a string of the form type/subtype,
        usable for a MIME Content-type header.
        The default implementation looks the file's extension
        up in the table self.extensions_map, using application/octet-stream
        as a default; however it would be permissible (if
        slow) to look inside the data to make a better guess.
        """

        base, ext = posixpath.splitext(path)
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        ext = ext.lower()
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        else:
            return self.extensions_map['']

    if not mimetypes.inited:
        mimetypes.init() # try to read system mime.types
    extensions_map = mimetypes.types_map.copy()
    extensions_map.update({
        '': 'application/octet-stream', # Default
        '.py': 'text/plain',
        '.c': 'text/plain',
        '.h': 'text/plain',
        })


def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = http.server.HTTPServer):
    http.server.test(HandlerClass, ServerClass)

if __name__ == '__main__':
test()
```

BaseHTTPServer – base classes for implementing web servers

https://pymotw.com/2/BaseHTTPServer/

(mieux : https://pymotw.com/3/http.server/)

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

Available In: 1.4 and later

`BaseHTTPServer` 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 your request handler class, implement the method `do_METHOD()`, replacing *METHOD* with the name of the HTTP method. For example, `do_GET()`, `do_POST()`, etc. For consistency, the method takes 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 which can be useful in building the response:

```
from BaseHTTPServer import BaseHTTPRequestHandler
import urlparse

class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        parsed_path = urlparse.urlparse(self.path)
        message_parts = [
                'CLIENT VALUES:',
                'client_address=%s (%s)' % (self.client_address,
                                            self.address_string()),
                'command=%s' % self.command,
                'path=%s' % self.path,
                'real path=%s' % parsed_path.path,
                'query=%s' % parsed_path.query,
                'request_version=%s' % self.request_version,
                '',
                'SERVER VALUES:',
                'server_version=%s' % self.server_version,
                'sys_version=%s' % self.sys_version,
                'protocol_version=%s' % self.protocol_version,
                '',
                'HEADERS RECEIVED:',
                ]
```

```
        for name, value in sorted(self.headers.items()):
            message_parts.append('%s=%s' % (name, value.rstrip()))
        message_parts.append('')
        message = '\r\n'.join(message_parts)
        self.send_response(200)
        self.end_headers()
        self.wfile.write(message)
        return

if __name__ == '__main__':
    from BaseHTTPServer 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:

```
$ python BaseHTTPServer_GET.py
Starting server, use <Ctrl-C> to stop
```

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

```
$ curl -i http://localhost:8080/?foo=barHTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 16:00:34 GMT

CLIENT VALUES:
client_address=('127.0.0.1', 51275) (localhost)
command=GET
path=/?foo=bar
real path=/
query=foo=bar
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

HTTP POST

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

```
from BaseHTTPServer import BaseHTTPRequestHandler
import cgi

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.end_headers()
        self.wfile.write('Client: %s\n' % str(self.client_address))
        self.wfile.write('User-agent: %s\n' % str(self.headers['user-agent']))
        self.wfile.write('Path: %s\n' % self.path)
        self.wfile.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
                self.wfile.write('\tUploaded %s as "%s" (%d bytes)\n' % \
                        (field, field_item.filename, file_len))
            else:
                # Regular form value
                self.wfile.write('\t%s=%s\n' % (field, form[field].value))
        return

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

curl can include form data in the message it posts to the server. The last argument, `-F datafile=@BaseHTTPServer_GET.py`, posts the contents of the file `BaseHTTPServer_GET.py` to illustrate reading file data from the form.

```
$ curl http://localhost:8080/ -F name=dhellmann -F foo=bar -F
datafile=@BaseHTTPServer_GET.py
Client: ('127.0.0.1', 51128)
Path: /
Form data:
        name=dhellmann
        foo=bar
        Uploaded datafile (2222 bytes)
```

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`.

```
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
from SocketServer import ThreadingMixIn
import threading

class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        message =  threading.currentThread().getName()
        self.wfile.write(message)
        self.wfile.write('\n')
        return

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()
```

Each time a request comes in, a new thread or process is created to handle it:

```
$ curl http://localhost:8080/
Thread-1
$ curl http://localhost:8080/
Thread-2
$ curl http://localhost:8080/
Thread-3
```

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

Handling Errors

Error handling is made easy with `send_error()`. Simply pass the appropriate error code and an optional error message, and the entire response (with headers, status code, and body) is generated automatically.

```
from BaseHTTPServer import BaseHTTPRequestHandler
```

```
class ErrorHandler(BaseHTTPRequestHandler):

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

if __name__ == '__main__':
    from BaseHTTPServer 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.

```
$ curl -i http://localhost:8080/
HTTP/1.0 404 Not Found
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 15:49:44 GMT
Content-Type: text/html
Connection: close

<head>
<title>Error response</title>
</head>
<body>
<h1>Error response</h1>
<p>Error code 404.
<p>Message: Not Found.
<p>Error code explanation: 404 = Nothing matches the given URI.
</body>
```

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.

```
from BaseHTTPServer import BaseHTTPRequestHandler
import urlparse
import time

class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.send_header('Last-Modified', self.date_time_string(time.time()))
        self.end_headers()
        self.wfile.write('Response body\n')
        return

if __name__ == '__main__':
    from BaseHTTPServer 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 2822**.

```
$ curl -i http://localhost:8080/
HTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.7
Date: Sun, 10 Oct 2010 13:58:32 GMT
Last-Modified: Sun, 10 Oct 2010 13:58:32 -0000

Response body
```

http.server — Base Classes for Implementing Web Servers

https://pymotw.com/3/http.server/