Bharat Chauhan

A semi-regular technical blog where I post tutorials or write about new things I learn.

Writing an HTTP server from scratch

Thu 16 November 2017 · updated Mon 19 October 2020 · Programming · python http server

Prelude ¶

This article will introduce you to the basics of HTTP servers and teach you how you can write one. We'll use the Python programming language to write our server.

Please note that this is a basic tutorial and, therefore, should not be taken as a guide for building a production level HTTP server.

GitHub repo ¶

You can find the code on Github and test it out: **https://github.com/bhch/crude-server**.

The code in the repo contains some extra modifications, but you'll figure those out when you read the source code.

Steps to create an HTTP server ¶

An HTTP server can be built in these two steps:

1. Create a TCP server ¶

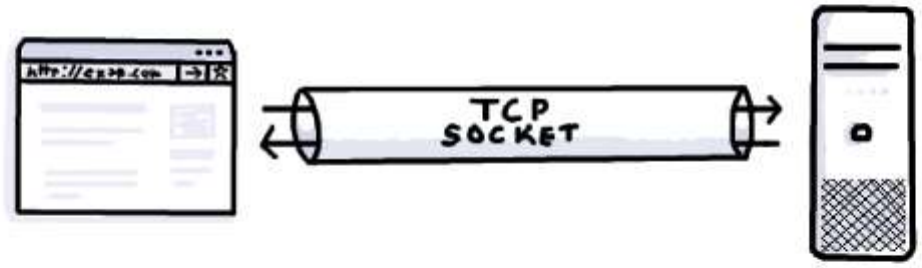
HTTP is built on top of TCP (Transmission Control Protocol), so we will need to create a program capable of sending and receiving data through a TCP socket. Knowledge about the TCP protocol is not required but familiarity with Python's `socket` library is. Go through `socket` library's docs, or read the PyMOTW page for good examples.

2. Teach our TCP server the HTTP protocol. ¶

At this point our TCP server doesn't understand HTTP. To convert it from a TCP server to an HTTP server all we need to do is *teach* it the HTTP protocol. You need to have some basic knowledge about the HTTP protocol, for example, the structure of HTTP requests and responses. A good starting point to learn about the HTTP protocol is Mozilla's docs on HTTP.

That's it. That's all it takes to write an HTTP server.

Step 1: Creating a TCP server ¶



HTTP requests and responses are sent through a TCP socket.

Since HTTP works through a TCP socket, we'll start by writing a simple TCP server. We will use Python's `socket` library for this.

1.2 A simple Echo server ¶

Firstly, let's create a simple version of our TCP server: an Echo server. An Echo server is a program that returns the data that it receives, nothing less, nothing more.

Start by creating a file called `main.py` for our code:

```
# main.py
import socket

class TCPServer:
    host = '127.0.0.1' # address for our server
    port = 8888 # port for our server

    def start(self):
        # create a socket object
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # bind the socket object to the address and port
        s.bind((self.host, self.port))
```

```
        # start listening for connections
        s.listen(5)

        print("Listening at", s.getsockname())

        while True:
            # accept any new connection
            conn, addr = s.accept()

            print("Connected by", addr)

            # read the data sent by the client
            # for the sake of this tutorial,
            # we'll only read the first 1024 bytes
            data = conn.recv(1024)

            # send back the data to client
            conn.sendall(data)

            # close the connection
            conn.close()

if __name__ == '__main__':
    server = TCPServer()
    server.start()
```

Read the comments in the code for explanation.

To test this server, we can create a TCP client but that won't be necessary. We can just use our web browser.

First, run the server like this:

```
$ python main.py
```

Now, visit `127.0.0.1:8888` from your browser. Your browser will send an HTTP request to our TCP server. Since our server is an Echo server, it will return the data that our browser sends it back to the browser.

You'll see a response from our server something like this:

```
GET / HTTP/1.1
Host: 127.0.0.1:8888
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 <...long string.
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Cookie: __utma=889955431.966891923.1514504807.15045637810 <...long string...>
```

1.3 Modifying the Echo server ¶

Our Echo server is very primitive and not very helpful. We'll now modify it such that it can be used as a base for our HTTP server.

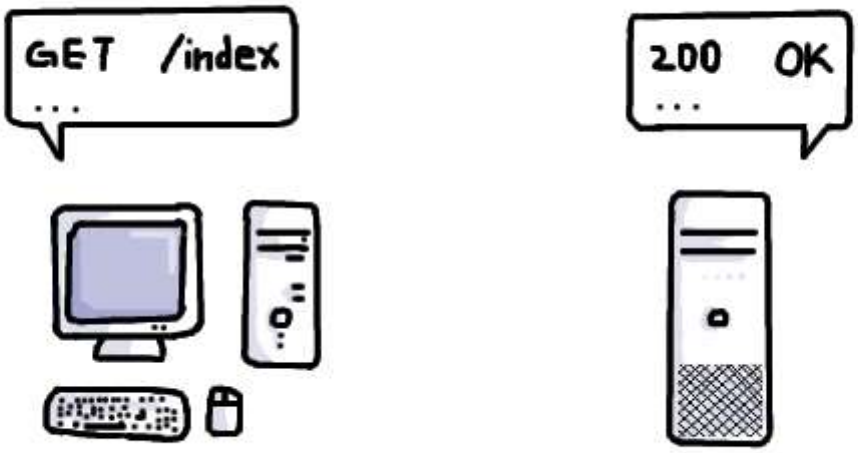
```
class TCPServer:
    def __init__(self, host='127.0.0.1', port=8888):
        self.host = host
        self.port = port

    def start(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((self.host, self.port))
        s.listen(5)

        print("Listening at", s.getsockname())

        while True:
            conn, addr = s.accept()
            print("Connected by", addr)
            data = conn.recv(1024)

            response = self.handle_request(data)

            conn.sendall(response)
            conn.close()

    def handle_request(self, data):
        """Handles incoming data and returns a response.
        Override this in subclass.
```

```
        """
        return data
```

The modifications we've done are:

1. We've created an `__init__` function that takes to arguments - `host` and `port`. We've set the default values for them as they were in the previous example. The reason we did this is now we can bind our server to different addresses and ports if we so desired.
2. We've also created a new method called `handle_request` which will come in handy when we subclass the `TCPServer` class. You might have noticed the code inside the `while` loop has also changed. The new code sends the data to `handle_request` method which will then returns a response. Currently, the `handle_request` method does nothing but we'll create a subclass and override this method to add some actual functionality.

Step 2: Teaching the HTTP protocol to our TCP server ¶



Instead of implementing the HTTP protocol in the `TCPServer` class, we'll subclass it and implement the protocol in the subclass. Let's start by handling the incoming requests by overriding the `handle_request` method. Add this to `main.py` file:

```
# main.py

# ... previous code ...

class HTTPServer(TCPServer):
    def handle_request(self, data):
        return b"Request received!" # send bytes, not string

if __name__ == '__main__':
    server = HTTPServer()
    server.start()
```

> **Important:** Python's `socket` library receives and sends data as `bytes`, not as `str` (string). That's why we're using the `b""` prefix with our strings. If we don't do that, we'll get an error.

Now **restart the server** by running the `main.py` file. Visit `127.0.0.1:8888` from your browser and you should see `Request received!` message on the screen.

Although, it's not a valid HTTP response yet. An HTTP response is made up of the following parts:

1. The response line a.k.a status line (it's the first line)
2. Response headers (optional)
3. A blank line
4. Response body (optional)

An example:

```
HTTP/1.1 200 OK              # The first line is called the response line
Server: Tornado/4.3          # Response header
Date: Wed, 18 Oct 2017       # Response header
Content-type: text/html      # Response header
Content-Length: 13           # Response header
                             # Blank line
Hello, world!                # Response body
```

The response line and the headers contain informaton for the browser. They are not shown to the client. The response body contains the data which the browser displays on the screen for the client.

The blank line plays a very important role in HTTP and many other protocols. It helps to seperate the headers and response body. Without it, there's no way for browsers to tell which part of the response should be shown to the user.

For a better understanding of HTTP responses, read the docs on Mozilla.

Let's make our server return a valid response this time:

```
class HTTPServer(TCPServer):
    def handle_request(self, data):
        response_line = b"HTTP/1.1 200 OK\r\n"

        blank_line = b"\r\n"

        response_body = r"Request received!""

        return b"".join([response_line, blank_line, response_body])
```

The `\r\n` characters are line break characters. They are present at the end of every line in an HTTP response except the response body. They are useful for browsers to tell headers and response line apart.

This time our server will return a valid HTTP response.

2.1 Headers ¶

Our server doesn't return any headers yet. Let's make it return some.

Headers contain some general information about the website and the current page, such as the name of the server, or the content type of the response, or any other information that might be useful for the browser but not so much for the user.

First, let's give our server a name. I'll call it `Crude Server` because it's crude, raw and a basic HTTP server.

Second, instead of returning plain text, we'll return HTML in the response. For that, we'll also have to send the `Content-Type` header to let the browser know that this is an HTML response.

Now, let's make our server return this information in headers:

```
class HTTPServer(TCPServer):
    def handle_request(self, data):
        response_line = b"HTTP/1.1 200 OK\r\n"

        headers = b"".join([
            b"Server: Crude Server\r\n",
            b"Content-Type: text/html\r\n"
        ])

        blank_line = b"\r\n"

        response_body = b"""<html>
            <body>
            <h1>Request received!</h1>
            <body>
            </html>
        """

        return b"".join([response_line, headers, blank_line, response_body])
```

2.2 Refining our HTTP server ¶

We are returning the HTTP response by manually writing it. It's not very convenient.

Let's create a few methods to compile response line, headers and response:

```
class HTTPServer(TCPServer):
    headers = {
        'Server': 'CrudeServer',
        'Content-Type': 'text/html',
    }

    status_codes = {
        200: 'OK',
        404: 'Not Found',
```

```
    }

    def handle_request(self, data):
        """Handles the incoming request.
        Compiles and returns the response
        """

        response_line = self.response_line(status_code=200)

        response_headers = self.response_headers()

        blank_line = b"\r\n"

        response_body = b"""
            <html>
                <body>
                    <h1>Request received!</h1>
                </body>
            </html>
        """

        return b"".join([response_line, response_headers, blank_line, respons

    def response_line(self, status_code):
        """Returns response line"""
        reason = self.status_codes[status_code]
        line = "HTTP/1.1 %s %s\r\n" % (status_code, reason)

        return line.encode() # call encode to convert str to bytes

    def response_headers(self, extra_headers=None):
        """Returns headers
        The `extra_headers` can be a dict for sending
        extra headers for the current response
        """
        headers_copy = self.headers.copy() # make a local copy of headers

        if extra_headers:
            headers_copy.update(extra_headers)

        headers = ""

        for h in headers_copy:
            headers += "%s: %s\r\n" % (h, headers_copy[h])

        return headers.encode() # call encode to convert str to bytes
```

So, we've created two new method: `response_line()` and `response_headers()` which will compile the response line and headers for us. The comments in the code will help clarify what's going on.

2.3 Status codes ¶

As you can see in the code above, we've created a dictionary of status codes in `HTTPServer` class. But there are only 2 status codes in our dictionary. I've done that intentionally to keep the code shorter. You can implement all the status codes in your local version of the code. See Mozilla's docs for a list of all the HTTP statuses.

You can use Python's `http.client` from the standard library. It has a dictionary of all HTTP statuses which can be accessed from `http.cleint.responses`.

2.4 HTTP requests ¶

Currently, our server just sends A response for every request. To make it more functional, we'll need to read the requests of the client and send them the content that they request.

Let's first understand what's an HTTP request.

An HTTP request is made up of the following parts:

1. The request line (it's the first line)
2. Request headers (optional)
3. A blank line
4. Request body (optional)

An example of typical HTTP request:

```
GET /index.html HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0
```

In case you haven't noticed, in the example above, the request doesn't have a body, just a Request line, headers and a blank line at the end (although, it's not visible here).

The most important part of an HTTP request is the Request line. Let's discuss this further below.

2.4.1 Request line ¶

The first line of an HTTP request is called the Request line. It consists of 4 parts:

```
   GET    /index.html      HTTP/1.1          \r\n
   \_/    \_________/      \______/          \____/
    |          |               |                |
 Method       URI        HTTP version       Line break
```

1. `Method` tells the server what action the client wants to perform on the `URI`. For example, `GET` method means the client wants the server to send him the page at the given `URI`. The `POST` method means the client wants to send some data to the server at the given `URI`.
2. `URI` stands for Uniform Resource Identifier. This tells the server on which resource or page or anything else the client wants to perform the request.
3. `HTTP version` is the version of HTTP the client supports or wants the server to use for the response. The most widely used version of HTTP is 1.1.
4. `Line break` - this tells the server that the request line has ended and the request headers follow after this line.

I've covered basics about the HTTP requests. You can read more about it here.

2.5 Parsing requests ¶

So far, we've rendered a response without actually *reading* the requests, which means our server is no good yet. We'll have to write some more code to parse the

incoming requests to know what HTTP method is used, to know what URI is requested, etc.

Let's write a simple request parser. We'll create a class called `HTTPRequest`. Add the following to the `main.py` file:

```
# main.py

# ...
# ... class TCPServer ...
# ...
# ... class HTTPServer ...
# ...


class HTTPRequest:
    def __init__(self, data):
        self.method = None
        self.uri = None
        self.http_version = "1.1" # default to HTTP/1.1 if request doesn't pr

        # call self.parse() method to parse the request data
        self.parse(data)

    def parse(self, data):
        lines = data.split(b"\r\n")

        request_line = lines[0]

        words = request_line.split(b" ")

        self.method = words[0].decode() # call decode to convert bytes to str

        if len(words) > 1:
            # we put this in an if-block because sometimes
            # browsers don't send uri for homepage
            self.uri = words[1].decode() # call decode to convert bytes to st

        if len(words) > 2:
            self.http_version = words[2]


# ...
# ... if __name__ == '__main__' ...
# ...
```

The above code only parses the Request line. We don't have to worry about request headers, or request body for this tutorial.

2.6 Implementing HTTP request methods ¶

Our server can't distinguish between different request methods. We are serving the same response for `GET` or `POST` or `OPTIONS` or any other request method.

We'll create one method in `HTTPServer` class—`handle_GET()` to handle `GET` requests.

For a `GET` request, a server returns the requested resource. Say, if a request comes in like `GET /index.html`, the server will send the contents of *index.html* file in the response body.

```
class HTTPServer(TCPServer):
    # ... other code ...

    def handle_request(self, data):
        # create an instance of `HTTPRequest`
        request = HTTPRequest(data)

        # now, look at the request method and call the
        # appropriate handler
        handler = getattr(self, 'handle_%s' % request.method)

        response = handler(request)

    return response

    def handle_GET(self, request):
        # We'll write this method a little later
        pass
```

Okay, we've created one handler to handle `GET` requests. In this tutoial, we won't implement handlers for other request types. But the Github repo contains handler for `OPTIONS` requests as well.

2.6.1 Sending 501 Response ¶

`501` HTTP responses mean `Not Implemented`. They are used for telling the client that the server does't support the requested method type.

Since our server will only handle `GET` requests, it's a good idea to send `501` response for other request methods such as `POST` or `PUT`.

So, let's modify our server to send a `501` response:

```
class HTTPServer(TCPServer):
    # ... headers ...

    status_codes = {
        200: 'OK',
        404: 'Not Found',
        501: 'Not Implemented',
    }

    def handle_request(self, data):
        request = HTTPRequest(data)

        try:
            handler = getattr(self, 'handle_%s' % request.method)
        except AttributeError:
            handler = self.HTTP_501_handler

        response = handler(request)

    return response

    def HTTP_501_handler(self, request):
        response_line = self.response_line(status_code=501)

        response_headers = self.response_headers()

        blank_line = b"\r\n"

        response_body = b"<h1>501 Not Implemented</h1>"

        return b"".join([response_line, response_headers, blank_line, respons

    def handle_GET(self, request):
        # We'll write this method a little later
        pass
```

Now, if a rquest comes in with any method other than `GET`, our server will appropriately return a `501` response.

2.7 Handling `GET` requests ¶

Let's start fully implementing the `handle_GET()` method. Now we'll serve content from actual HTML and other files.

2.7.1 Serving content from files ¶

Let's serve some content from files instead of from strings as we've been doing until now.

First, let's create a couple of files - *"index.html"* and *"hello.html"*.

Now, if a request comes in like `GET /index.html`, we'll serve the *index.html* file. If a request comes in like `GET /hello.html`, we'll serve the *hello.html* file.

Keep both these files in the same directory as `main.py`.

index.html:

```
<html>
    <head>
        <title>Index page</title>
    </head>
    <body>
        <h1>Index page</h1>
        <p>This is the index page.</p>
    </body>
</html>
```

hello.html:

```
<html>
    <head>
        <title>Hello page</title>
    </head>
    <body>
        <h1>Hello page</h1>
        <p>This is the hello page.</p>
    </body>
</html>
```

We'll also send a `404 Not Found` response if a requested file doesn't exist.

Now, let's modify the handler method to serve content according to the request URI.

```
import os # remember to import os
import socket


class HTTPServer(TCPServer):
    # ... other code ...

    def handle_GET(self, request):
        filename = request.uri.strip('/') # remove the slash from the request

        if os.path.exists(filename):
            response_line = self.response_line(status_code=200)

            response_headers = self.response_headers()

            with open(filename, 'rb') as f:
                response_body = f.read()
        else:
            response_line = self.response_line(status_code=404)
            response_headers = self.response_headers()
            response_body = b"<h1>404 Not Found</h1>"

        blank_line = b"\r\n"

        return b"".join([response_line, response_headers, blank_line, respons
```

Now, **restart the server** and visit `127.0.0.1:8888/index.html` and you should see the response containing the contents of *index.html* file.

Also, try and visit a random url like `127.0.0.1:8888/blahblah.html` and you should get the `404 Not Found` response.

2.8 Serving images and other media ¶

To serve images, or videos, or any other media a server must send a proper `Content-Type` header to the browser to tell that the browser about what kind of content is being served so it can render the content appropriately.

Our server, as it is Eventually, can serve images, or videos. You can even try it - put an image file next to the previously created HTML files and make a request for that image from the browser - `127.0.0.1:8888/picture.png` and you should see some random gibberish text on your screen. This is happening because our server is sending the image with `Content-Type: text/html` header. That is why the browser is not rendering it as an image.

The `Content-Type` of a file is also called MIME Type. We'll be using Python's `mimetypes` library to know a file's MIME type from it's extension. We can do it manually, but there are hundreds of different file types - *"jpg, png, mp4, mp3, svg ...*", so, why not use a library to make it easier.

`mimetypes` library is included in Python's standard library, so there's no need to install anything.

Let's modify our server to send proper `Content-Type` header for different files.

```
import os
import socket
import mimetypes # remember to import mimetypes


class HTTPServer(TCPServer):
    # ... other code ...

    def handle_GET(self, request):
        filename = request.uri.strip('/')

        if os.path.exists(filename):
```

```
            response_line = self.response_line(status_code=200)

            # find out a file's MIME type
            # if nothing is found, just send `text/html`
            content_type = mimetypes.guess_type(filename)[0] or 'text/html'

            extra_headers = {'Content-Type': content_type}
            response_headers = self.response_headers(extra_headers)

        # ... rest of the code ...
```

Now, put an image next to *index.html*.

Restart the server and make a request for it from your browser.

You should see the image rendered properly. You can also try and embed the image in *index.html* file using `<img src="image-name.png">` and then request the `/index.html` page. You should see the image there, too.

And it's done ¶

We've built our server! Well, at least a minimal server for learning purpose. Now you can play around with it—try and implement more request methods, or more headers.

Also, see the code in the supplementary Github repo. It contains some imporovements to the code, more comments and information.

Hopefully, the way HTTP and HTTP servers work is clear to you now.

Taking our server to the next level ¶

We've only scratched the surface of the HTTP protocol and it's workings. And the server written in this tutorial is far from good. But if you want to improve the server you can see the following checkpoints that almost every production level server implements:

1. **Implement full HTTP**:

   Our server only supports one method—`GET`—and a few headers. It has no support for Cookies, or caching. A good starting point towards a production ready server would be to implement the full HTTP protocol. You can start by reading Mozilla's web docs and implementing all the HTTP features one by one.
2. **Serving multiple clients**:

   Currently, our server only serves one client at a time. You can make it serve more than one client at a time by multi-threading. But not all servers use multi-threading. There are event driven servers which use a different approach. Read about the Tornado web server.
3. **WSGI support**:

   Currently, Python frameworks like Django and Flask won't work with our server. The only way to make our server work with them is by implementing WSGI. What is WSGI? Here's a tutorial by Armin Ronacher.
4. **Daemon mode**:

   Running a server in daemon mode means that running it as a background process. Currently, our server doesn't run in background. So, if we run it and then close the terminal, our server process will also stop. You can use tools like supervisord to make the server run in daemon mode.
5. **Logging**:

   Logging is a very import feature of servers. It means writing logs to a file in the event of say, an error occurs, like a `404` error, or a `500` server error. Servers also write access logs for every request which contain information about the IP Address of the the client, the page requested and the time.
6. **Configuration support**:

   Our server doesn't support configuration, for example we can't tell our server which files or directories to server, where to find files, etc. This way, we can map a file to a different URL, which means we can serve `hello.html` file from a URL like `/goodbye`.
7. **Serving large files**:

   Our server is capable of serving videos. But not very large videos. The reason is before serving files to the client we read the whole file in the memory by using `f.read()` (see the code in `handle_GET` method above). So, if you try to serve large video file, say about 3 GB, and you only have 2 GB of RAM, your server

will crash. The way servers safely serve large files is by reading and writing only a smart part (around 10 - 20 MB) of the file at a time. This is done in a loop which will run until the whole file is read and written to the response. This technique is called *chunking*.

8. **Security**:
   In this blog post, we've talked about almost everything HTTP related, except for security. This is a very delicate topic and it's never permanent - you'll have to constantly keep working to keep your code secure. A good starting point to read about server security would be this page by Mozilla. But the Most Unexceptional way to keep a software secure is by making it open source. That way if someone finds a security issue, they might notify you or even write a patch themselves. Why do you think almost every web server is open source? This is why.

That's it ¶

You now know about the internal workings of an HTTP server.

In case there are errors in the code, or any other mistakes, please let me know.

Okyanus • 2 years ago

can you publish the code on GitHub?

1 • Reply • Share ›

> **bhch** Author → Okyanus • a year ago
>
> Hello, I've posted the code on github: https://github.com/bhch/cru... Thanks for suggesting that.
>
> Disqus didn't send me any emails about new comments so, I apologise for the late reply.
>
> • Reply • Share ›

Balkar Singh • a year ago

Hello Sir

I had tried using your code on python 3.7 and there were lot of errors.
Most of them i had resolved on my own but i was still not able to display the image on the webpage. Can you help in it an also explain why it was't being displayed.

Thanks
Balkar Singh

• Reply • Share ›

> **bhch** Author → Balkar Singh • a year ago
>
> Ah, sorry about that. It was written for Python 2.7. I've now updated this post. The code is also available on github: https://github.com/bhch/cru...
>
> Disqus didn't send me any emails about new comments so, I apologise for the late reply.
>
> • Reply • Share ›