João Ventura FOLLOW
Python enthusiast with 10 years of experience!

Building a basic HTTP Server from scratch in Python

Published Nov 22, 2020

In its essence, the modern web is just text going back and forth between clients and servers. As developers, we often use web frameworks to help us build strings to send to the clients. Web frameworks abstract us from the underlying "textual reality" by parsing the incoming http requests (which is just a string), call the corresponding function, and build a string response (mostly by using templates). Finally, clients parse those strings and do whatever they want from it.

This blog post shows how to build a barebones HTTP server from scratch and it is based on an exercise I gave to my MSc students. The only pre-requisite is a basic understanding of Python 3. If you want to implement this as we go along, you can grab the starting application from this link. The final source code can be found in this gist.

HTTP is just text

HTTP is the protocol that browsers use to retrieve and push information to servers. In its essence HTTP is just text that follows a certain pattern: on the first line you specify which resource you want, then it follows the headers, and then you have a blank line that separates the headers from the body of the message (if any). Here's how you could retrieve the about page of a website:

```
GET /about.html HTTP/1.0
User-Agent:
```

Enjoy this post? ♡ 9 3

And here's how you could send some form data to a web server, using the POST method:

```
POST /form.php HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

name=John&surname=Doe
```

To prove that this is only text, you could simply copy-paste the text above and use something that allows you to send text over a network. Let's use telnet to get the google's about page:

```
$> telnet google.com 80
Trying 84.91.171.170...
Connected to google.com.
Escape character is '^]'.

(1)
GET /about/ HTTP/1.0

(2)
HTTP/1.0 200 OK
Vary: Accept-Encoding
Content-Type: text/html
Date: Thu, 09 Feb 2017 16:41:37 GMT
Expires: Thu, 09 Feb 2017 16:41:37 GMT
Cache-Control: private, max-age=0
Last-Modified: Thu, 08 Dec 2016 01:00:57 GMT
X-Content-Type-Options: nosniff
Server: sffe
X-XSS-Protection: 1; mode=block
Accept-Ranges: none

<!DOCTYPE html>
    <html class="google mmfb" lang="en">
        <head>
        ...

</html>
Connection closed by foreign host.
```

```
$> telnet httpbin.org 80

Trying 54.175.219.8...
Connected to httpbin.org.
Escape character is '^]'.

(1)
POST /post HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

name=John&surname=Doe

(2)
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 09 Feb 2017 16:38:26 GMT
Content-Type: application/json
Content-Length: 328
Connection: close
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

{
  "form": {
    "name": "John",
    "surname": "Doe"
  },
  "headers": {
    "Content-Length": "21",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org"
  },
  "url": "http://httpbin.org/post"
}
Connection closed by foreign host.
```

You can see the HTTP requests **(1)** followed by the HTTP server responses in **(2)**. Same pattern on requests and responses, and text everywhere! More information about HTTP on the excellent High Performance Browser Networking book.

If you are planning to implement network applications from scratch, you'll probably need to work with network sockets. A socket is an abstraction provided by your operating system that allows you to send and receive bytes through a network. Here's a basic implementation of an HTTP server (you can get it from this link):

```
"""
 Implements a simple HTTP/1.0 Server

"""


import socket



# Define socket host and port
SERVER_HOST = '0.0.0.0'
SERVER_PORT = 8000

# Create socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((SERVER_HOST, SERVER_PORT))
server_socket.listen(1)
print('Listening on port %s ...' % SERVER_PORT)

while True:
    # Wait for client connections
    client_connection, client_address = server_socket.accept()

    # Get the client request
    request = client_connection.recv(1024).decode()
    print(request)

    # Send HTTP response
    response = 'HTTP/1.0 200 OK\n\nHello World'
    client_connection.sendall(response.encode())
    client_connection.close()

# Close socket
server_socket.close()
```

We start by de variable and

there to set up the socket to listen for requests on the given (host, port). Check the Python docs on sockets for more info.

The rest of the code is self-explanatory: wait for client connections, read the request string, send an HTTP-formatted string with *Hello World* on the response body and close the client connection. We do this forever (or until someone presses Ctrl+C). Open your browser on http://localhost:8000/ and you should see the server's response:

As an exercise, change the *Hello World* to *<h1>Hello World</h1>* and see what happens. And did you see the

in the

```
Listening on port 8000 ...

GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:51.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-PT,pt;q=0.8,en;q=0.5,en-US;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

Yes, it's the browser requesting the root page ("/") of the server..

Index.html

By default, when a browser requests the root of a server (using an HTTP request such as **GET / HTTP/1.0**), we should return the *index.html* page. Let's change the code inside the while to always return the contents of the *htdocs/index.html* file.

```
while True:
    # Wait for client connections
    (...)
    # Get the client request
    (...)

    # Get the content of htdocs/index.html
    fin = open('htdocs/index.html')
    content = fin.read()
    fin.close()

    # Send HTTP response
    response = 'HTTP/1.0 200 OK\n\n' + content
    client_connection.sendall(response.encode())
    (...)
```

Basically, we read the contents of the file and add it to the *response* string as message body, instead of the previous *Hello World*. The *index.html* file is just a text file (inside the *htdocs* directory) with

```
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h1>Hello World!</h1>
    <p>Welcome to the index.html web page..</p>
    <p>Here's a link to <a href="ipsum.html">Ipsum</a></p>
</body>
</html>
```

Here's how it should look like in the browser:

You can click on the link as many times as you want, but you server will **always** return the contents of *index.html*. It was programmed to behave that way!

Return other pages

So far our server returns the *index.html* page but we should allow it to return other pages as well. Technically, it means that we must parse the first line of the HTTP request (which is something like *GET /ipsum.html HTTP/1.0*), open the intended file and returns its contents. Here's the changes:

```
while True:
    # Wait for client connections
    (...)
    # Get the client request
    (...)

    # Parse HTTP headers
    headers = request.split('\n')
    filename = headers[0].split()[1]

    # Get the content of the file
    if filename == '/':
        filename = '/index.html'

    fin = open('htdocs' + filename)
    content = fin.read()
    fin.close()

    # Send HTTP response
    response = 'HTTP/1.0 200 OK\n\n' + content
    client_connection.sendall(response.encode())
    (...)
```

Basically, we extract the filename from the request string, open the file (we assume that all html files are inside the **htdocs** folder) and return its content. You can also check that we correctly return the *index.html* file when the clients ask for the root resource ('/').

Here's the content of *htdocs/ipsum.html*:

```
<html>
<head>
    <title>Ipsum</title>
</head>
<body>
    <h1>Ipsum!</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Pellentesque tincidunt libero diam, nec imperdiet libero
        sodales quis. Nulla in pulvinar sem. Vivamus placerat
        ullamcorper sagittis. Proin varius, erat sed egestas semper,
        enim lectus viverra diam, id placerat est augue et turpis.
    </p>
</body>
</html>
```

Try it on your own code, and see if you can open the **index.html** and **ipsum.html** files.

404 - Not found

We're not done yet! This is what happens if we try to request a file that does not exist, such as http://localhost:8000/hello.html:

```
GET /hello.html HTTP/1.1

Traceback (most recent call last):
  File "httpserver.py", line 36, in <module>
    fin = open('htdocs' + filename)
FileNotFoundError: [Errno 2] No such file or directory: 'htdocs/hello.html'
```

The server crashes and exits!

We need to catch the exception and return a 404 response:

```
while True:
    # Wait for client connections
    (...)
    # Get the client request
    (...)

    # Parse HTTP headers
    headers = request.split('\n')
    filename = headers[0].split()[1]

    # Get the content of the file
    if filename == '/':
        filename = '/index.html'

    try:
        fin = open('htdocs' + filename)
        content = fin.read()
        fin.close()

        response = 'HTTP/1.0 200 OK\n\n' + content
    except FileNotFoundError:

        response = 'HTTP/1.0 404 NOT FOUND\n\nFile Not Found'

    # Send HTTP response
    client_connection.sendall(response.encode())
    client_connection.close()
```

If you want, you can change the body of the Http 404 response to have personalized error messages..

The entire source code for this example can be found in this gist.

Python Server side Distributed systems

Enjoy this post? Give **João Ventura** a like if it's helpful.

♡ 9 3 SHARE

João Ventura

Python enthusiast with 10 years of experience!

I'm João Ventura, a software engineer from Portugal. Currently I work as a Computer Science Professor at the Polytechnique Institute of Setúbal, where I teach the Operating Systems and Distributed Computing courses. Some of my ar...

FOLLOW

3 Replies

Leave a reply

A **avensis david** 7 months ago

very interesting thanks for sharing

Enjoy this post? ♡ 9 3

Muhammad Abdullah 7 months ago

How are you

♡ Reply

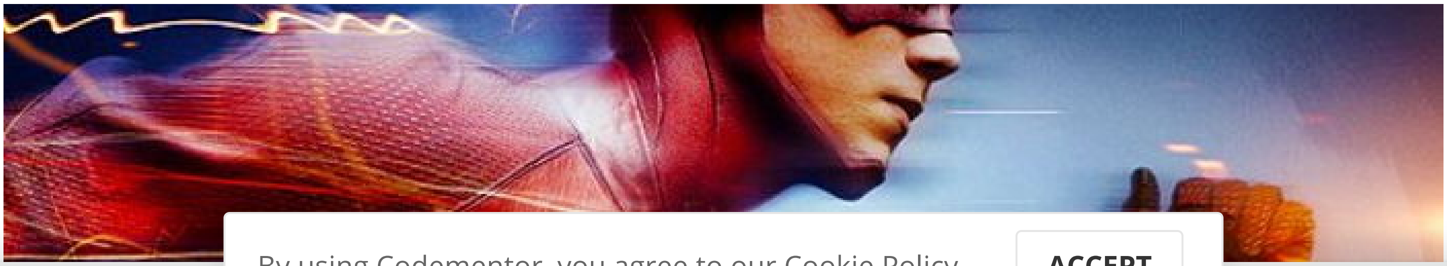
Muhammad Abdullah 7 months ago

Hello

♡ Reply

Olaide Alaka

TEST DJANGO APPS QUICKLY



Enjoy this post? ♡ 9 3

I'm sure you agree that you have to check to see that your codes do what you expect them to do.
Manually testing your code may come as default...

- You go through the user interface and imitate the processes users are expected to follow.
- You use some "print" statements to check inputs and outputs.
- Where a process depends on some database items, you create dummy objects.
- After all these, you ship your application...phew!
  ...you think you finally finished the project!
  Well most of the time, you thought wrong! It is either a bug snuck through or a new feature needs to be added!
- You w ...

READ MORE