Lab: Building a Simple Web Server[1]

Objective

The web is just a sequence of texts going back and forth between clients and servers. As developers, we can use web frameworks to build messages to send to the clients. Web frameworks abstract the underlying "textual content" by parsing the incoming http requests (which is just a sequence of strings), calling the corresponding function and building a response (which is also a sequence of strings). Finally, clients parse those strings and do whatever they want from it. Note that Python has provided HTTPServer class to support web services. However, this lab illustrates how to build a basic HTTP server from scratch.

The purpose of this lab is to:

- introduce the framework of HTTP server programming using Python socket;
- develop a simple Web server.

Tasks

HTTP

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.html” page from a website:

```
GET /about.html HTTP/1.1
User-Agent: Mozilla/5.0
```

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

```
POST /form.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

name=Jack&surname=Chan
```

[1] Reference: Joao Ventura, Building a basic HTTP server from scratch in Python, https://www.codementor.io/@joaojonesventura/building-a-basic-http-server-from-scratch-in-python-1cedkg0842

Sending HTTP Responses Using Sockets

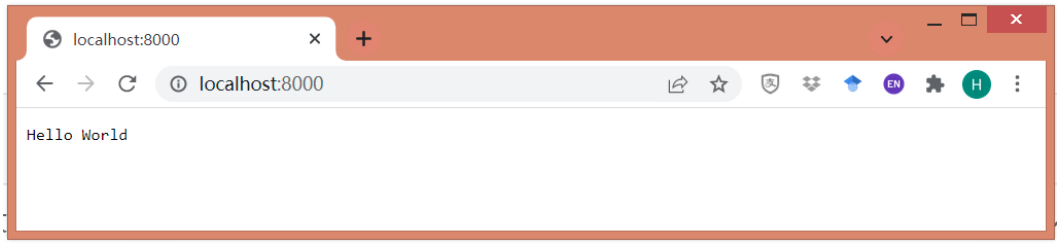
If you are planning to implement network applications from scratch, you'll need to work with network sockets. A socket is an abstraction provided by your operating system that allows you to send and receive bytes through a network. Here's a basic implementation of an HTTP server:

```
# Implements a simple HTTP 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/1.1 200 OK\n\nHello World'
    client_connection.sendall(response.encode())
    client_connection.close()

# Close socket
server_socket.close()
```

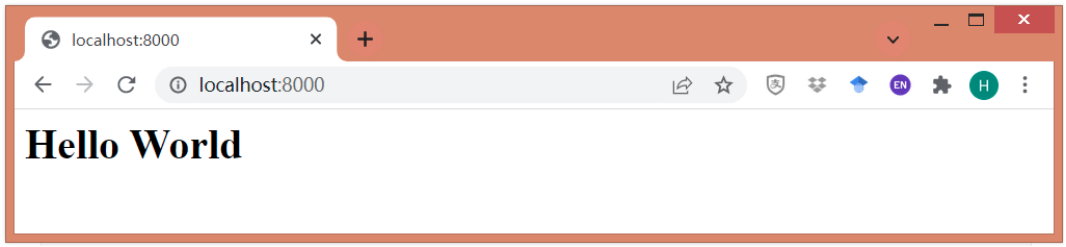
Recall the previous lab about socket programming. We start by defining the server's socket host and port. Then, we create the server_socket variable and set it to AF_INET (IPv4 address family) and SOCK_STREAM (TCP socket). The server_socket is listening for requests on the given address (host, port). Then the server creates a socket connection to the client, reads the request string from the socket connection, sends an HTTP-formatted string

with “Hello World” text on the response body and closes the socket connection. The server does this forever (or until someone presses Ctrl+C).

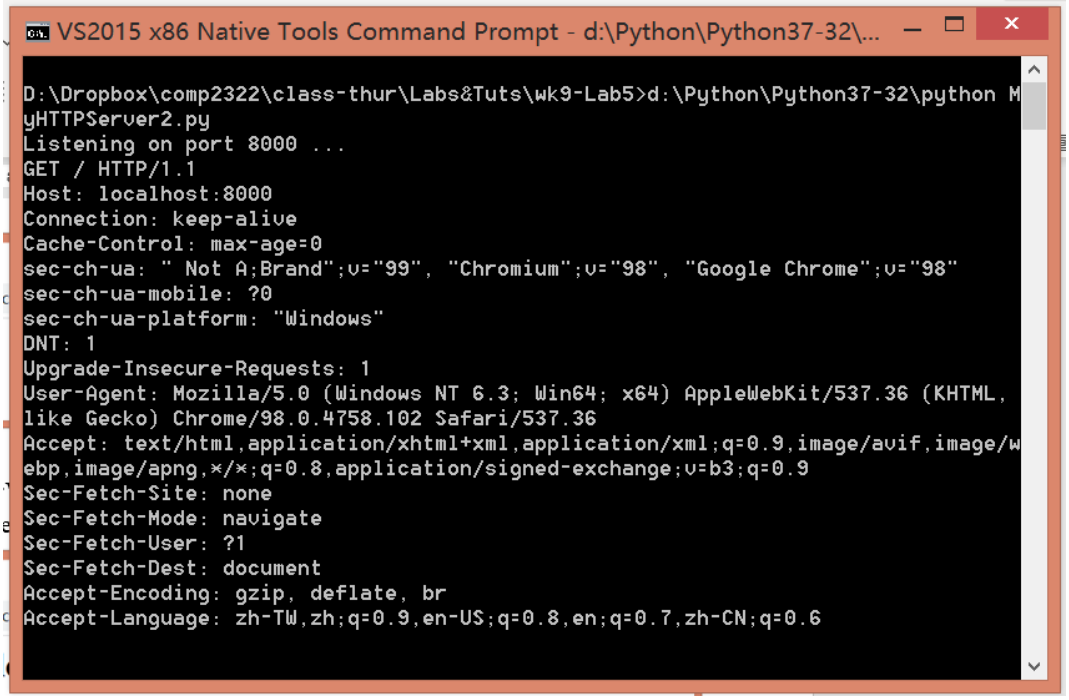
Now save the above Python program as “MyHTTPServer1.py” in your working directory and run it from the window console. Open your browser with URL “http://localhost:8000/” and you should see the server's response on the browser:



Change the “Hello World” text to the “<h1>Hello World</h1>” text in the above Python program, save it as “MyHTTPServer2.py” in your working directory, and run it from the window console. Open your browser with URL “http://localhost:8000/” and you will see the following response on the browser:



And you can see the outputs in the server program’s window console:



Yes, it is the browser requesting the root web page ("/") of the server.

Default Web Page: Index.html

By default, when a browser requests the root of a server (using an HTTP request such as GET / HTTP/1.1), the server should return the “index.html” page. We need to change the code inside the while loop of the server program 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” text. The “index.html” file is just a text file (inside the htdocs directory) with the following html content:

```
<html>
<head>
    <link rel="icon" href="data:,">
    <title>Index</title>
</head>
<body>
    <h1>Welcome to the index.html web page.</h1>
    <p>Here's a link to <a href="helloworld.html">hello world</a>.</p>
</body>
</html>
```

Note that when the browser makes an HTTP request for a html file from the server, it also automatically generates another HTTP request for a “favicon.ico” file in the same location. To avoid this, we can add a line in the <head> section in the “index.html”:

```
    <link rel="icon" href="data:,">
```

This line assigns an empty data URL to the favicon’s element which specifies the location of the external resource. This trick stops the user’s browser from sending an automatic HTTP

request for the favicon.

Now save the above Python program as “MyHTTPServer3.py” in your working directory, and run it from the window console. Open your browser with URL “http://localhost:8000/” and it should look like this in the browser:

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

Return Other Pages

So far the 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 /helloworld.html HTTP/1.1), open the intended file and return its contents.

Here are the changes in the Python code:

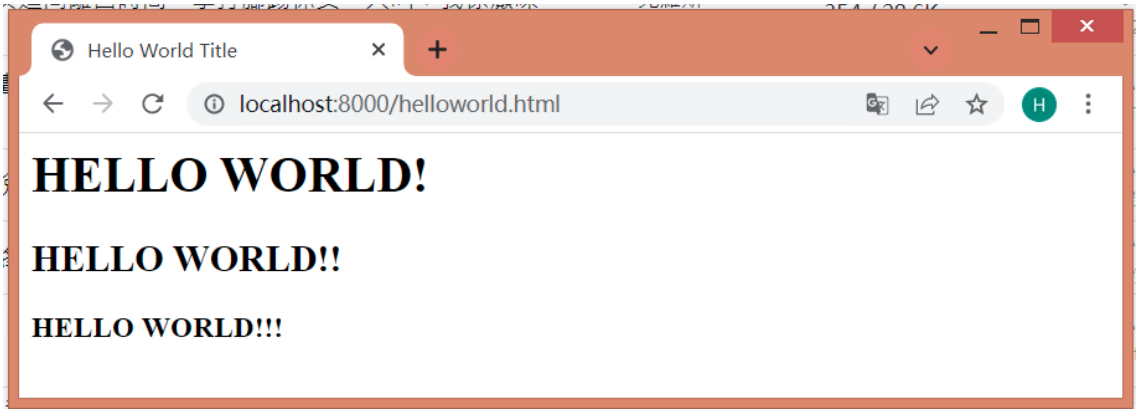
```
while True:
    # Wait for client connections
    (...)
    # Get the client request
    (...)

    # 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
    (...)
```

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 client asks for the root ('/').

Here's the content of the “htdocs/helloworld.html” file:

```
<html>
<head>
    <link rel="icon" href="data:,">
    <title>Hello World Title</title>
</head>
<body>
        <h1>HELLO WORLD!</h1>
        <h2>HELLO WORLD!!</h2>
        <h3>HELLO WORLD!!!</h3>
</body>
</html>
```

Save the above Python program as “MyHTTPServer4.py” in your working directory, and run it from the window console. You will see how the browser displays with URL “http://localhost:8000/helloworld.html”:



File Not Found

We are not done yet! If we try to request a file that does not exist, such as http://localhost:8000/hello.html, you can see in the console that the server program crashes!

```
GET /hello.html HTTP/1.1

Traceback (most recent call last):
  File "MyHTTPServer4.py", line 31, in <module>
    fin = open('htdocs' + filename)
FileNotFoundError: [Errno 2] No such file or directory: 'htdocs/hello.html'
```

We need to catch the exception and return a 404 response if the requested file does not exist in the server:

```
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/1.1 200 OK\n\n' + content
    except FileNotFoundError:
        response = 'HTTP/1.1 404 Not Found\n\nFile Not Found'

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

Save the above Python program as “MyHTTPServer5.py” in your working directory, and run it from the window console. You will see how the browser displays with URL “http://localhost:8000/hello.html”:

Parse the HTTP Request Type

So far the server program does not parse the HTTP request type and always assumes the request is a GET command. We can add some codes to make the serve program first parse the command type when it receives the client request. If the command type is "GET", it further process the GET command; otherwise, it returns a "400 Bad Request" response.

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

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

    # Parse the request type
    if request_type == 'GET':
        # process the GET request
        (…)
    else:
        response = 'HTTP/1.1 400 Bad Request\n\nRequest Not Supported'

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

The complete source code of the server program is shown as follow:

```
# Implements a simple HTTP Server
import socket

# Handle the HTTP request.
def handle_request(request):

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

```
    # Parse the request type
    if request_type == 'GET':
        # Get the content of the file
        if filename == '/':
            filename = '/index.html'

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

            response = 'HTTP/1.1 200 OK\n\n' + content
        except FileNotFoundError:
            response = 'HTTP/1.1 404 Not Found\n\nFile Not Found'
    else:
        response = 'HTTP/1.1 400 Bad Request\n\nRequest Not Supported'

    return response

# Define socket host and port
SERVER_HOST = 'localhost'
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:\n')
    print(request)

    # Send HTTP response
    response = handle_request(request)
    client_connection.sendall(response.encode())

    # Close connection
    client_connection.close()
```

```
# Close socket
server_socket.close()
```

You can use the following client program to see how the server program responds to the GET command, e.g., trying the commands “GET /helloworld.html HTTP:/1.1” or “HEAD /helloworld.html HTTP:/1.1”:

```
# Implements a simple HTTP client
import socket

SERVER_HOST = '127.0.0.1'
SERVER_PORT = 8000

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((SERVER_HOST, SERVER_PORT))
request = input('Input HTTP request command:\n')
client_socket.send(request.encode())
response = client_socket.recv(1024)
print ('Server response:\n')
print (response.decode())
client_socket.close()
```

You now have successfully built a simple HTTP server socket program on your computer from scratch. This server program can respond properly to the several HTTP requests. More sophisticated HTTP server that can respond to different types of HTTP commands can be developed based on this program.