© Nicola Lombardi, Hardware and Software Engineer

www.linkedin.com/in/nicola-lombardi-09046b205/

Overview

REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. This Server had been implemented with RESTFul principles: Client– Server, Uniform Interfaces, Stateless, Cacheble and Layered System.

The Python script managing the server functionality involves several object classes focused on handling HTTP requests and managing multiple client connections.

1. **BaseHTTPRequestHandler**: This class, from the BaseHTTPServer module, processes incoming HTTP requests. However, it cannot respond to requests on its own, as it needs methods to handle them. The myHandler class ***extends*** BaseHTTPRequestHandler, inheriting its methods, demonstrating object-oriented programming principles like inheritance. A key method used is path, which retrieves the resource's path from the request. This REST WebServerHTTP had been **Overridden**.

2. **ThreadingServer**: This class, created by combining ThreadingMixIn and HTTPServer, enables the server to handle multiple concurrent connections (multicast). It uses the constructor method, accepting parameters like myHandler, server_name, and port_name to set up the server. The class itself has no attributes or methods, except for a placeholder pass statement.

3. **HTTPServer**: This class, which ThreadingServer extends, builds on the TCPServer class. Before introducing multithreading, servers were constructed using HTTPServer methods, requiring the same parameters (server name, port, and handler) to manage client requests.

In summary, the script showcases inheritance and multithreading in Python to manage concurrent HTTP requests on the server.

Requirements

You need to install:

- Python 2.8
- Lubuntu or Ubuntu or Linux (Ubuntu 15.04 is enough)
- Two LX terminal windows

The server is executed by running a Python script via the Linux terminal command `python webServerOOP_overridden.py` in an lxterminal window (a Linux command prompt that allows interaction with the operating system). This starts the server, which listens on the specified port number and remains active, ready to receive HTTP requests until it is manually interrupted. These requests will be processed by the methods defined in the script's class. Requests are sent to the server's listening address using a different terminal window with the Linux curl command.

`curl` is a command-line tool used to transfer data to or from a server using supported protocols such as DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, and more.

There are many ways to use curl, depending on the protocol and task at hand. Here, I will focus on the command syntax for using the HTTP protocol. With this command, you can send requests to the server, where the terminal window hosting these curl commands acts as the client's user-agent (the interface for communicating with the web server).

The basic syntax is simple: type curl, followed by the `-X method_name` flag to specify the HTTP method the server should use to process the request, then provide the server address in the format localhost:port_number/resource_path. The default method is GET, so if the user types curl localhost:port_number/resource_path, the server will respond using the GET method.

To use the POST method, the command follows the standard syntax: `curl -X POST localhost:port_number/resource_path`, along with parameters specified with -d, e.g., `-d username=Nicola -d password=password.`

For the HEAD method, there are three options:

- `curl -i localhost:port_number/resource_path`: Sends a GET request and returns both the header and the body, effectively combining a GET and HEAD response.
- `curl -I localhost:port_number/resource_path`: Sends a HEAD request and returns only the status line, server version, HTTP protocol version, and current date/time.
- `curl -X HEAD localhost:port_number/resource_path`: Sends a pure HEAD request, returning all headers related to the server, client, and specified resource.

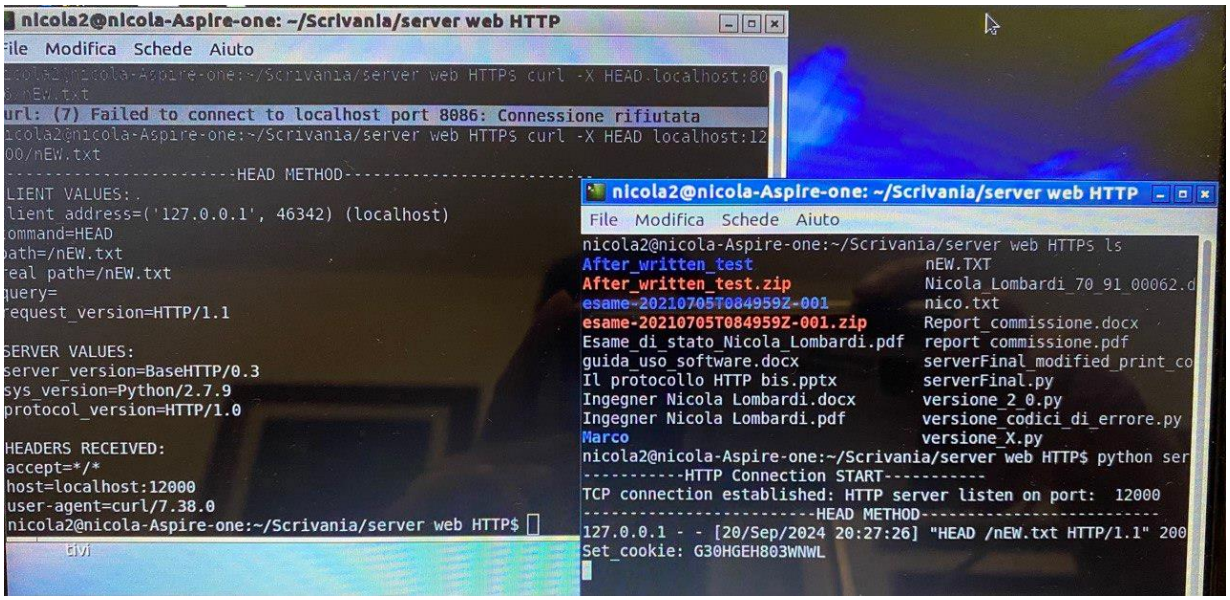

Fig.1 : Example of Curl in Linux

Simple Running using the Firefox web browser

Using Linux OS, the script reaches <192.168.1.43 : 1200 > with HTTP connection:

Fig.2: Example of HTTP response in the Client view (2nd LX terminal window)

In the Linux Shell there is the resource index.html :

```
nicola2@nicola-Aspire-one:~$ curl 192.168.1.20:12000
<html>
<head><title>HTTP Server</title>
<h1><center>Homepage HTTP server 1.1</center></h1></head>
<body bgcolor="#E6E6FA">
        Hai appena usufruito del metodo <b>GET</b> del protocollo:<br><br><br>
        <img src="find-http.jpg">
        <p>
                Con il metodo GET di http puoi:
                <ul>
                  <li>richiedere una risorsa (recuperare le informazioni localiz
zate dal Request-URI)</li>
                  <li>inviare dei dati in modo non sicuro (per questo esiste il
metodo <i>POST</i>)</li>
                  <li>richiedere una risorsa con le modifiche richieste nei camp
i If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, o If-Range</l
i>
                </ul>

        </p>

        <p>     In questo server trattero' solo il metodo GET nella sua forma pu
```

Fig.2: Example of HTTP response in the Client view (2nd LX terminal window)

By initializing the variable sendReply to false, it is indicated that the MIME type of the resource the client is requesting has not yet been recognized. The format analysis is performed using self.path.endswith, which considers only the part of the URL after the last period, i.e., the format such as 'txt'. The following figure shows these code instructions:

```
if sendReply:
        #
        # opening of file (you kwnow the path inside URL) required and send
        #
        f = open(curdir + sep + self.path)
        self.send_response(200)
        self.send_header('Content-type',mimetype)
        self.end_headers()
        self.wfile.write(f.read())
        f.close()
        import os
        st = os.stat(self.path[1:100])
```

Fig.3: if the Mimetype exists

If the format matches one of those present in the conditional statements, the corresponding MIME type is identified, and the boolean variable is assigned the value true. If sendReply is equal to true, the file is then opened.

The file is opened using the open() Linux System Call C-language like method, to which the modules curdir, sep, and the string self.path are passed as parameters to correctly read the file that needs to be opened and subsequently returned together with the response headers.

If we wish to retrieve any file, it will be sufficient to connect to the correct IP address (if from the local host that hosts the server, then simply 'localhost', which corresponds to 127.0.0.1), specify the correct path, and the file will be accessible, regardless of its format!

Finally, the HTTP response is constructed by calculating the content length of the resource in bytes (content-length), the calculation of which is performed using the stat method from the os library, taking into account the result provided by path starting from the second character (at position 1) of the string, as self.path returns "/path".

The GET response headers will include the status line and other standard fields.

Regarding the data to be sent via HTTP POST, the first step is to determine the length of the data to be transmitted to the server. In the code, no specific length is defined, which implies the possibility of sending an unlimited number of parameters. The parameter string is stored in the variable post_data once the server reads the incoming data. This method returns the HTTP response containing the data in the payload field. Unlike the traditional HTTP POST method typically used by a form, this method merely sends the data and confirms receipt by returning these parameters as a string.

Here is the code block corresponding to the method:

Fig.3: Example of HTTP POST Overridden method

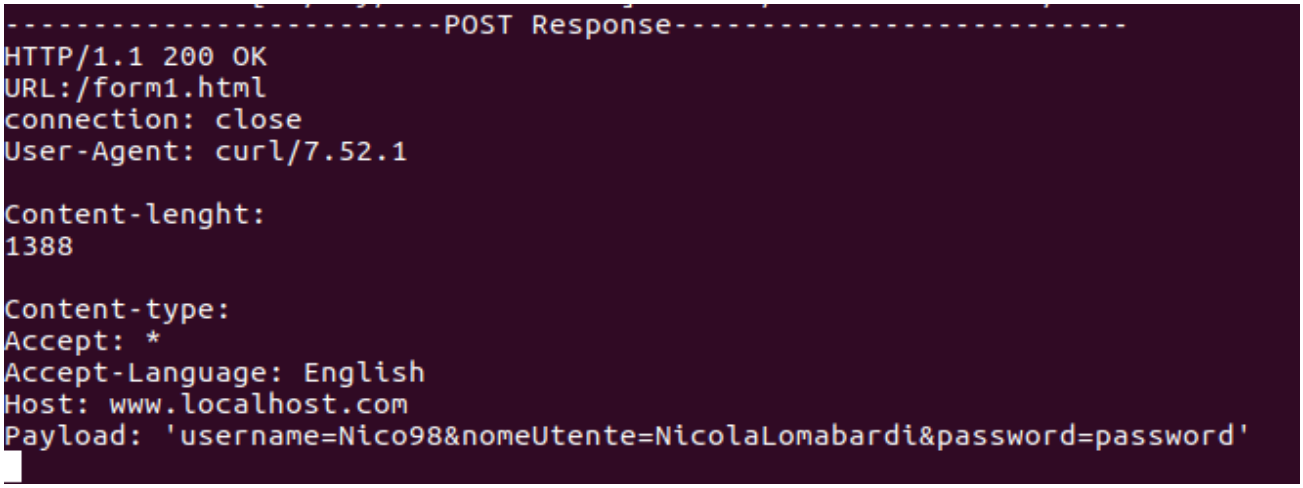

Fig.4: POST response

Other Methods:

- Head

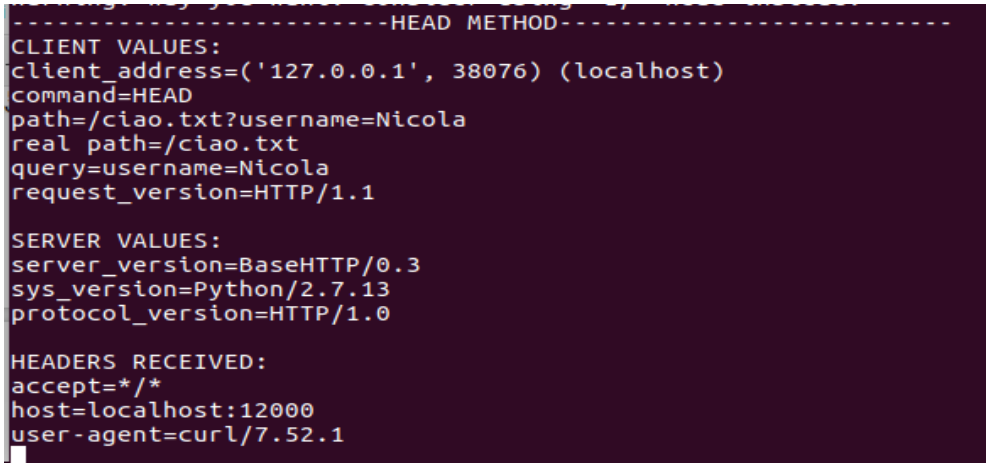

Fig.5: HEAD response