

Relazione di progetto

Corso di Programmazione di Reti / A.A 2023 - 2024

Federico Bagattoni

Federico Bagattoni – 1070857

federico.bagattoni@studio.unibo.it

Corso di Laurea in Ingegneria e Scienze Informatiche

Alma Mater Studiorum – Università di Bologna



Sommario

1 – Richiesta	2
2 – Implementazione	3
3 – Funzionamento	5
3.1 – Requisiti	5
3.2 – Avvio ed utilizzo del server	5
3.3 – Test di altre richieste non supportate	5

1 – Richiesta


Traccia 2: Web Server Semplice

Il progetto ha come scopo la realizzazione di un semplice *web server* in linguaggio Python che possa servire file statici (come HTML, CSS ed immagini) e gestire richieste HTTP GET di base. Il server deve essere in grado di gestire più richieste **simultaneamente** e restituire risposte appropriate ai client.

2 – Implementazione

L'intero progetto è disponibile presso la repository <https://github.com/bagarozzi/NP23-http>. L'implementazione delle web server si basa sull'apertura di un `ThreadingTCPServer` che sarà responsabile di gestire richieste da più client simultaneamente dedicando un `Thread` separato per ogni client.

Prima di avviare il server vengono impostati i parametri `allow_reuse_address` e `daemon_threads`; il primo permette di riutilizzare la stessa socket senza aspettare che il kernel l'abbia rilasciata mentre il secondo dichiara che i thread creati per la gestione delle connessioni ai singoli client siano demoni, quindi autonomi, ed in caso di chiusura del processo loro termineranno assieme ad esso. Così inoltre non è necessario che il main aspetti la terminazione di ogni thread.



```

50 def main():
51     port = SERVER_DEFAULT_PORT
52     if sys.argv[1:]:
53         port = int(sys.argv[1:])
54     server_socket = ('localhost', port)
55
56     signal.signal(signal.SIGINT, terminationSignalHandler)
57
58     try:
59         server = socketserver.ThreadingTCPServer(server_socket, GetRequestHandler)
60         server.allow_reuse_address = True
61         server.daemon_threads = True
62         print("Server listening...")
63         server.serve_forever()
64     except KeyboardInterrupt:
65         server.shutdown()
66
67 if __name__ == "__main__":
68     main()

```

Immagine 2.1 – La funzione main

Al costruttore di `ThreadingTCPServer` viene passata una classe personalizzata chiamata `GetRequestHandler`. Il server chiamerà i metodi di questa classe per gestire le varie richieste HTTP che gli vengono fatte.

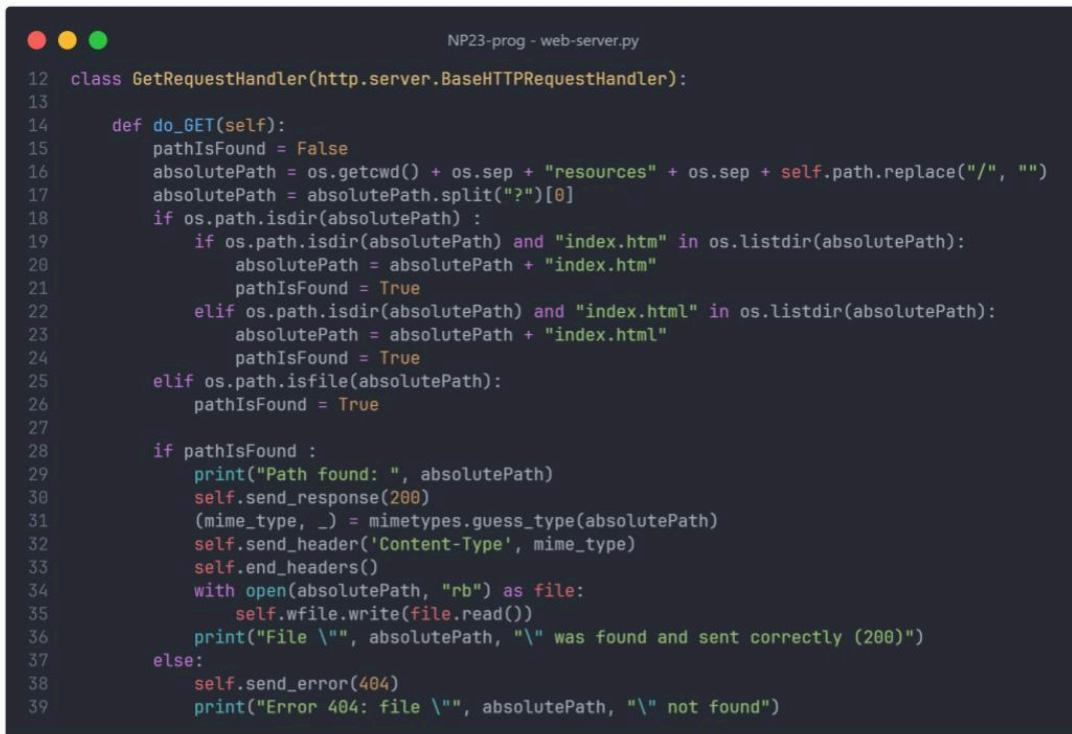
Tale classe estende `BaseHTTPRequestHandler`, un'interfaccia che mette a disposizione vari metodi per la gestione delle richieste HTTP, tra cui: `do_GET()`, `do_POST()` etc...

Visto che lo scopo dell'elaborato è quello di servire le richieste GET, di questa interfaccia viene implementato solo il metodo `do_GET()`.

All'interno del metodo `do_GET()` viene preso il percorso richiesto e viene formattato correttamente rimuovendo le query e muovendo il percorso nella directory "resources", che funge da directory di lavoro per le risorse che deve inviare il server.

Successivamente vengono fatte diverse operazioni distinte a seconda del percorso richiesto:

- se il percorso richiesto è una directory vengono cercati, ed eventualmente inviati, i file `index.html` e `index.htm` (in questo ordine)
- se il percorso richiesto è un file esistente viene inviato
- se nessuna delle due precedenti condizioni è raggiunta, allora viene restituita una pagina di errore con `response code 404`



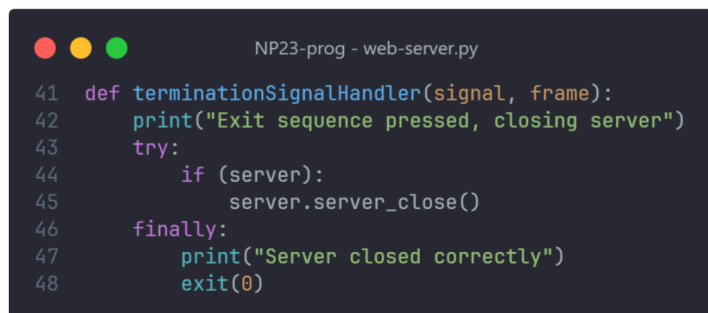
```

12 class GetRequestHandler(http.server.BaseHTTPRequestHandler):
13
14     def do_GET(self):
15         pathIsFound = False
16         absolutePath = os.getcwd() + os.sep + "resources" + os.sep + self.path.replace("/", "")
17         absolutePath = absolutePath.split("?")[0]
18         if os.path.isdir(absolutePath):
19             if os.path.isdir(absolutePath) and "index.htm" in os.listdir(absolutePath):
20                 absolutePath = absolutePath + "index.htm"
21                 pathIsFound = True
22             elif os.path.isdir(absolutePath) and "index.html" in os.listdir(absolutePath):
23                 absolutePath = absolutePath + "index.html"
24                 pathIsFound = True
25         elif os.path.isfile(absolutePath):
26             pathIsFound = True
27
28         if pathIsFound:
29             print("Path found: ", absolutePath)
30             self.send_response(200)
31             (mime_type, _) = mimetypes.guess_type(absolutePath)
32             self.send_header('Content-Type', mime_type)
33             self.end_headers()
34             with open(absolutePath, "rb") as file:
35                 self.wfile.write(file.read())
36             print("File \"", absolutePath, "\" was found and sent correctly (200)")
37         else:
38             self.send_error(404)
39             print("Error 404: file \"", absolutePath, "\" not found")

```

Immagine 2.2 – L'implementazione della classe GetRequestHandler

Allo script è stato anche aggiunto un metodo per la corretta chiusura del server, infatti quando viene premuta la sequenza di terminazione `CTRL + C` chiama il metodo di chiusura del server.



```

41 def terminationSignalHandler(signal, frame):
42     print("Exit sequence pressed, closing server")
43     try:
44         if (server):
45             server.server_close()
46     finally:
47         print("Server closed correctly")
48         exit(0)

```

Immagine 2.3 – Metodo per la gestione della sequenza di terminazione

3 – Funzionamento

3.1 – Requisiti

Il funzionamento dello script richiede la versione di Python 3.9.2 o superiori.

3.2 – Avvio ed utilizzo del server

Il server viene avviato come un normale script python, dopo essersi posizionati nella root directory del progetto. Opzionalmente si può passare il numero di porta che si vuole che il server utilizzi, in alternativa verrà usato il numero di porta predefinito 8080.

```
user@unibo:~$ python web-server.py [port]
```

Successivamente l'utente può eseguire le richieste dal proprio browser, collegandosi all'indirizzo seguente, inserendo la porta passata in riga di comando o 8080 se non si ha passato nessun argomento:

```
http://localhost:[porta]
```

Una volta avviato si potrà vedere dalla linea di comando il log del server in tempo reale:

```
PS C:\Users\feder\Documents\NP23-prog> python web-server.py
Server listening...
Path found: C:\Users\feder\Documents\NP23-prog\resources\index.html
127.0.0.1 - - [04/Jun/2024 14:08:57] "GET / HTTP/1.1" 200 -
File "C:\Users\feder\Documents\NP23-prog\resources\index.html" was found and sent correctly (200)
Path found: C:\Users\feder\Documents\NP23-prog\resources\horse.jpeg
127.0.0.1 - - [04/Jun/2024 14:08:57] "GET /horse.jpeg HTTP/1.1" 200 -
File "C:\Users\feder\Documents\NP23-prog\resources\horse.jpeg" was found and sent correctly (200)
127.0.0.1 - - [04/Jun/2024 14:09:14] code 404, message Not Found
127.0.0.1 - - [04/Jun/2024 14:09:14] "GET /nonesiste HTTP/1.1" 404 -
Error 404: file "C:\Users\feder\Documents\NP23-prog\resources\nonesiste" not found
```

Immagine 3.2.1 – Output del web server gestendo varie richieste GET

Osserviamo come alla richiesta della pagina `index.html` il client richiede la pagina e la risorsa `horse.jpeg` in due richieste separate che vengono entrambe completate, mentre la richiesta della directory `nonesiste` viene corrisposta con un `404` in quanto la directory non esiste o non contiene i file `index.html` o `index.htm`.

3.3 – Test di altre richieste non supportate

Come menzionato nel capitolo 2, il server supporta solo richieste GET. Al fine di verificare che le richieste di altro tipo non siano supportate ho realizzato uno script che esegue

richieste HEAD e POST utilizzando il pacchetto requests. Lo si può trovare nella repository sotto il nome di `test.py`.

Lo script richiede che il server sia stato precedentemente avviato ed eventualmente deve essere avviato passando come primo ed unico argomento la porta con cui si è avviato il server; se nessun argomento gli è passato allora userà la porta predefinita come il server 8080.

```
user@unibo:~$ python test.py [port]
```

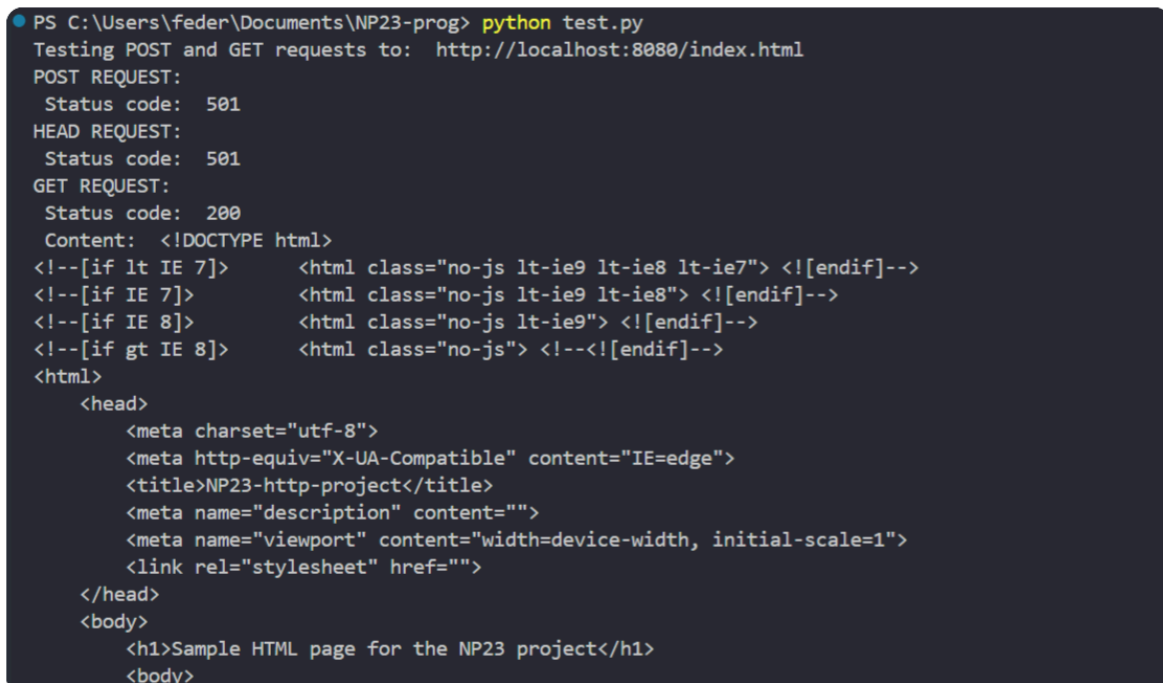
Successivamente lo script eseguirà richieste POST, GET ed HEAD verso l'indirizzo del server richiedendo la pagina `index.html` e stampando il risultato a video.



```
NP23-prog - test.py
10 address = "http://localhost:" + str(port) + "/index.html"
11
12 print("Testing POST and GET requests to: ", address)
13
14 try:
15     get_request = requests.get(address)
16     post_request = requests.post(address)
17     head_request = requests.head(address)
18 except requests.exceptions.Exception :
19     print("Exception in making the test requests, make sure the server is running and has something to display")
```

Immagine 3.3.1 – Richieste eseguite dallo script `test.py`

Se tutto è corretto l'output dello script è il seguente:



```
PS C:\Users\feder\Documents\NP23-prog> python test.py
Testing POST and GET requests to: http://localhost:8080/index.html
POST REQUEST:
Status code: 501
HEAD REQUEST:
Status code: 501
GET REQUEST:
Status code: 200
Content: <!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]> <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]> <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]> <html class="no-js"> <!--<![endif]-->
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>NP23-http-project</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="">
  </head>
  <body>
    <h1>Sample HTML page for the NP23 project</h1>
  </body>
```

Immagine 3.3.2 – Output dello script `test.py`

Possiamo osservare che le richieste POST ed HEAD vengono corrisposte con codice 501, Not Implemented, questo significa che il server non supporta la funzionalità richiesta

quindi il test è verificato.