

Documentazione per Progetto di Programmazione di Reti

Michele Farneti: 0001080677

July 2024

Contents

1	Introduzione	2
1.1	Generalità studente	2
1.2	Traccia	2
2	Guida all'uso	3
2.1	Avvio	3
2.2	Esecuzione	3
3	Analisi del codice del server	5
3.1	Librerie Utilizzate	5
3.2	Setting di porta ed indirizzo	6
3.3	Creazione ed impostazione del server	7
3.4	Gestione della chiusura	8
4	Analisi delle risposte HTTP più frequenti	9

Chapter 1

Introduzione

1.1 Generalità studente

- Studente: Michele Farneti
- Matricola: 0001080677
- Email: michele.farneti@studio.unibo.it

1.2 Traccia

Traccia 2: Creare un web server semplice in Python che possa servire file statici (come HTML, CSS, 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.

Chapter 2

Guida all'uso

2.1 Avvio

Il Web Server può essere avviato in due distinte maniere partendo dal file ***server.py***, situato nella root folder del progetto. Nel primo caso senza specificare argomenti a linea del comando: tale operazione porterà all'avvio del server http che verrà inizializzato con un indirizzo standard e si metterà in ascolto su una porta predefinita. La coppia indirizzo:porta standard è stata impostata come: ***localhost:8080***. Nel caso in cui sia invece desiderato che il server venga messo in ascolto su una porta differente, sarà sufficiente avviare l'esecuzione del file specificando come unico argomento la porta desiderata.

Avvio su porta standard: `python ./server.py`

Avvio su porta specifica: `python server.py 8081`

2.2 Esecuzione

Una volta avviata l'esecuzione del codice, qualora l'inizializzazione del server avvenga con successo, verrà mostrata a riga di comando una stringa riportante il successo dell'operazione, nonché il link con cui effettuare la prima richiesta http al server: Sarà sufficiente eseguire ***Ctrl + Click*** su di esso per eseguirla.

Da questo momento sarà possibile navigare il web server (anche da più client contemporaneamente) visualizzando da linea di comando informazioni riguardo ciascuna richiesta di tipo **GET** effettuata al server HTTP. Viene riportato di seguito un esempio:

```
127.0.0.1 - - [15/Jul/2024 22:53:29] "GET / HTTP/1.1" 304 -  
Request (Type: GET for path: /index.html) received from... 127.0.0.1:54022
```

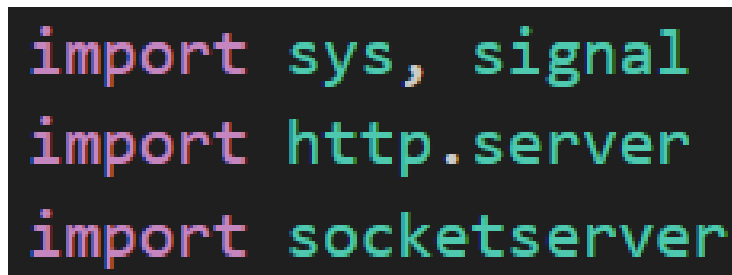
Figure 2.1: Le due righe stampate per ogni richiesta

Dalla prima riga è possibile ricavare il tipo della richiesta (**GET**), il path nel file system del server per la risorsa desiderata, nonché indirizzo e porta da cui si sta interfacciando il client. Nel caso in cui non fosse specificato una risorsa specifica (/) la richiesta sarà automaticamente reindirizzata su *index.html*. Dalla seconda riga invece è possibile ottenere altre informazioni riguardo la richiesta quali la data e soprattutto il *codice di risposta restituito dal server HTTP*. Qualora si desiderasse chiudere il server, sarà sufficiente utilizzare da linea di comando la sequenza di tasti **CTRL + C**. Viene garantita una chiusura sicura con annessa corretta terminazione dei thread per la gestione dei client.

Chapter 3

Analisi del codice del server

3.1 Librerie Utilizzate



```
import sys, signal
import http.server
import socketserver
```

Figure 3.1: Import nel codice

- `sys`: Modulo `sys`, in questo caso utilizzato sia per ricevere argomenti da riga di comando sia per terminare dell'applicazione una volta chiuso il server.
- `signal`: Tale modulo ci permette di realizzare handlers personalizzati. Nel nostro specifico caso l'handler sarà realizzato in modo da comandare la chiusura del server e conseguentemente dell'applicazione in caso di pressione di *CTRL+C*.
- `http.server`: Fornisce classi per realizzare server HTTP di base.
- `socketserver`: Fornisce varie classi che possono semplificare l'implementazione di server multi-threaded.

3.2 Setting di porta ed indirizzo

```
STANDARD_ADDRESS = "localhost"
STANDARD_PORT = 8080

if(len(sys.argv) == 1):
    PORT = STANDARD_PORT
else:
    PORT = int(sys.argv[1])
```

Figure 3.2: Setting di porta ed indirizzo

In questa sezione del codice del server vengono inizializzati i valori per indirizzo e porta, riferiti al Server, verso i quali bisognerà interfacciarsi per effettuare le richieste HTTP. In particolare l'host è settato come "**localhost**", indirizzo riservato che sta ad indicare l'IP della macchina corrente, solitamente corrisponde a *127.0.0.1*. Per quanto concerne la porta invece, è per convenzione impostata al valore standard *8080*, viene tuttavia permesso di cambiare tale valore specificando a riga di comando come unico argomento il nuovo numero di porta che desideriamo impostare.

3.3 Creazione ed impostazione del server

```
class HTTPHandler(http.server.SimpleHTTPRequestHandler):
    def do_GET(self):
        path = self.path
        client_ip, client_port = self.client_address
        command = self.command
        print(f"Request (Type: {command} for path: {self.path})
              received from... {client_ip}:{client_port}")
        return super().do_GET()
server = socketserver.ThreadingTCPServer((STANDARD_ADDRESS,PORT),HTTPHandler)
```

Figure 3.3: Creazione del server

Essendo il server realizzato leggero e con contenuti statici semplici quali pagine HTML e fogli di stile css, andiamo a sfruttare il modulo `http.server` per realizzare l'**event handler**. In particolare sono andato a creare una nuova sottoclasse *HTTPHandler* che andasse ad estendere la classe *http.server.SimpleHTTPRequestHandler*, così da sovrascrivere il metodo "do_Get()" in modo tale da aggiungere una stampa personalizzata al momento della gestione della richiesta HTTP. Entrando nei particolari, l'handler *SimpleHTTPRequestHandler* fornisce i file dalla directory corrente e di tutte quelle inferiori, mappando la struttura delle directory alle richieste HTTP. Implementa due funzioni che sono `do_HEAD()` e `do_GET()`, la seconda in particolare è colei che si occupa di mappare nel file system le richieste di GET. Inoltre è anche possibile mappare la richiesta su una directory specifica lasciando che essa venga automaticamente scansionata alla ricerca di un file `index.html`

La classe *ThreadingTCPServer* utilizzata per la realizzazione del server, permette la gestione contemporanea di più richieste fornendo dunque un servente di tipologia **multithread** in grado di gestire parallelamente i diversi client per mezzo della creazione di processi figli incaricati di rispondere.

Per il corretto funzionamento del server è inoltre necessario impostare a *true* l'attributo **allow_reuse_address** del server, indicando che può riutilizzare la stessa porta senza dover attendere che il kernel rilasci la porta sottostante, impostazione non consentita di default.

Una volta inizializzato il server viene dunque chiamata la sua funzione **serv_forever**, che gestisce le richieste finché il programma non termina o finché non viene esplicitamente interrotto.

3.4 Gestione della chiusura

```
def signal_handler(signal, frame):
    print( "Exiting http server (Ctrl+C pressed)")
    try:
        if( server ):
            server.server_close()
    finally:
        sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
```

Figure 3.4: Realizzazione handler chiusura

Il meccanismo di chiusura è realizzato sfruttando il modulo `signal`, con cui viene realizzato un handler personalizzato che, alla pressione della sequenza CTRL + C, interrompe il loop infinito che circonda `server.serve_forever()`, andando poi a chiudere il server ed a terminare l'applicazione. E' tuttavia necessario aver impostato a true *`server.daemon_threads`* per assicurarsi la corretta chiusura di tutti i thread alla chiusura del server.

```
try:
    while True:
        print(f"Server set up on Address: http://{STANDARD_ADDRESS}:{PORT}")
        print("Server waiting for requests...")
        server.serve_forever()
except KeyboardInterrupt:
    pass
```

Figure 3.5: Loop da interrompere

Chapter 4

Analisi delle risposte HTTP più frequenti

Come detto in precedenza al lancio dell'applicazione sarà possibile visualizzare su console informazioni riguardo le richieste ricevute dal server HTTP e alle relative risposte. Vengono di seguito analizzati i codici di stato più frequentemente riportati durante la navigazione delle pagine HTML:

200 : **OK**, il server è riuscito con successo a rintracciare la pagina richiesta nella directory e viene dunque restituita correttamente la risorsa.

404 : **NOT FOUND**, la risorsa richiesta non è risultata rintracciabile, viene dunque visualizzata una pagina di errore. Nell'`index.html` è stato implementando un bottone ad-oc con reindirizzamento ad una pagina inesistente per permettere di visualizzare tale codice a console.

304 : **NOT MODIFIED**, la risorsa richiesta, per mezzo degli header ***If-Modified-Since*** o ***If-None-Match***, viene identificata come già a disposizione del client e immutata dall'ultima trasmissione. Non risulta quindi necessario ritrasmetterla.