Documentazione per Progetto di Programmazione di Reti

Michele Farneti: 0001080677

July 2025

Contents

Chapter 1

Introduzione

1.1 Generalità studente

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

1.2 Traccia

Traccia 1 – Realizzazione di un Web Server minimale in Python e pubblicazione di un sito statico: Progettare un semplice server HTTP in Python (usando socket) e servire un sito web statico con HTML/CSS.

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, nonchè 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 il log delle informazioni riguardo ciascuna richiesta di tipo **GET** effettuata al server HTTP. Viene riportato di seguito un esempio:

Figure 2.1: Le due righe stampate per ogni richiesta

La **prima riga del log** è gestita manualmente all'interno del metodo `do_GET()` e fornisce informazioni dettagliate sulla richiesta ricevuta: è possibile identificare il **tipo di richiesta** (ad esempio `GET`), la **risorsa richiesta** (come `/index.html`) e l'indirizzo IP e la porta del **client** che ha effettuato la richiesta. Se la richiesta non specifica esplicitamente una risorsa (ad esempio `/`), il server tenterà automaticamente di servire la pagina `index.html`, se disponibile. La **seconda riga del log**, invece, è generata automaticamente dalla classe base `SimpleHTTPRequestHandler` e riporta in formato standard HTTP:

- la **data e ora** della richiesta,
- il **metodo HTTP** utilizzato,
- la **risorsa richiesta**,
- il **codice di risposta** restituito dal server.

Per **interrompere** in modo sicuro l'esecuzione del server, è sufficiente premere la combinazione di tasti `CTRL + C` nella console. Il server è configurato per gestire questo evento correttamente, garantendo la **chiusura ordinata** e la **terminazione sicura dei thread** eventualmente attivi per la gestione dei client.

Chapter 3

Analisi del codice del server

3.1 Librerie Utilizzate

```
import sys, signal
import http.server
import socketserver
import os
from datetime import datetime
```

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.
- os: Permette l'interazione con il sistema operativo. In questo progetto è utilizzato per cambiare la directory di lavoro del server (`os.chdir()`) e per verificare l'esistenza di file richiesti dal client.
- datetime: Utilizzato per ottenere la data e ora correnti nel momento in cui viene gestita una richiesta HTTP, al fine di includerle nei log personalizzati.

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

```python
class HTTPHandler(http.server.SimpleHTTPRequestHandler):
    def do_GET(self):
        path = self.path
        client_ip, client_port = self.client_address
        print(f"\r\n==> [{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] Richiesta  "+
              "( Tipo = GET per la risorsa : {self.path} ) ricevuta da {client_ip}:{client_port}")

        # Gestione speciale per documentazione.pdf
        if path == "/documentazione.pdf":
            file_path = "." + path  # assuming current directory root
            if not os.path.exists(file_path):
                self.send_response(404)
                self.send_header("Content-type", "text/html")
                self.end_headers()
                self.wfile.write(b"<h1>404 Not Found</h1><p>Il file documentazione.pdf non esiste.</p>")
                return

        return super().do_GET()

    def send_error(self, code, message=None, explain=None):
        if code == 404:
            self.send_response(404)
            self.send_header("Content-type", "text/html")
            self.end_headers()
            try:
                with open("404.html", "rb") as f:
                    self.wfile.write(f.read())
            except FileNotFoundError:
                self.wfile.write(b"<h1>404 Not Found</h1><p>La pagina richiesta non esiste.</p>")
        else:
            super().send_error(code, message, explain)
```

Figure 3.3: Creazione del server

Essendo il server realizzato leggero e con contenuti statici semplici, quali pagine HTML e fogli di stile CSS, si è deciso di sfruttare il modulo `http.server` per realizzare l'**event handler**. In particolare, è stata creata una nuova sottoclasse `HTTPHandler` che estende la classe `http.server.SimpleHTTPRequestHandler`, così da poter sovrascrivere il metodo `do_GET()` e aggiungere una stampa personalizzata nel momento in cui viene gestita una richiesta HTTP.

Nel dettaglio, il metodo `do_GET()` è stato modificato per stampare nel terminale un log dettagliato della richiesta, comprensivo di data, tipo, risorsa richiesta, indirizzo IP e porta del client. È stata inoltre aggiunta una gestione specifica per la risorsa `documentazione.pdf`: qualora tale file non fosse presente nella directory del server, verrà restituito un errore `404` con una risposta HTML dedicata, evitando così che la gestione della richiesta venga delegata al metodo originale.

Oltre a ciò, è stato sovrascritto anche il metodo `send_error()` per personalizzare la risposta in caso di errore `404`. In questo caso, il server tenta di restituire un file `404.html` personalizzato; qualora anche questo non fosse presente, viene inviata una risposta HTML generica con un messaggio di errore.

L'handler `SimpleHTTPRequestHandler` fornisce i file a partire dalla directory corrente e da tutte quelle inferiori, mappando la struttura del file system alle richieste HTTP ricevute. Tra i metodi disponibili, `do_GET()` è quello che gestisce direttamente le richieste di tipo GET, traducendole nel recupero della risorsa corrispondente nel file system. Se la richiesta punta a una directory, il server tenterà automaticamente di restituire un file `index.html` se presente.

La classe `ThreadingTCPServer`, utilizzata per la realizzazione del server, consente la gestione concorrente di più richieste, fornendo un servente di tipo **multithread** in grado di gestire in parallelo più client grazie alla creazione di thread separati per ciascuna connessione.

Per il corretto funzionamento del server è inoltre necessario impostare a `True` l'attributo `allow_reuse_address`, che consente al server di riutilizzare la stessa porta senza dover attendere il rilascio della risorsa da parte del sistema operativo, cosa non abilitata di default.

Una volta inizializzato, il server viene avviato tramite la chiamata al metodo `serve_forever()`, che entra in un ciclo di attesa e gestione delle richieste fino a quando l'esecuzione non viene esplicitamente interrotta. Questo meccanismo è gestito in modo sicuro: alla pressione di `CTRL+C`, viene invocato un *signal handler* che provvede alla chiusura ordinata del server e alla terminazione corretta dei thread ancora attivi.

3.4 Gestione dei MIME types

Gestione dei MIME types

Nel contesto della gestione dei contenuti statici, un aspetto fondamentale è rappresentato dalla corretta impostazione dei *MIME types* (Multipurpose Internet Mail Extensions), i quali permettono al browser del client di comprendere la tipologia di contenuto ricevuto e trattarlo di conseguenza (es. visualizzarlo, scaricarlo o aprirlo con un'applicazione dedicata).

Nel caso dell'implementazione sviluppata, la gestione dei MIME types è demandata al modulo `http.server`, che include un mapping automatico dei principali tipi di file: ad esempio, richieste a risorse come `index.html` o `stile.css` verranno servite con il corretto header `Content-Type` rispettivamente `text/html` e `text/css`.

Un'attenzione particolare è stata dedicata a file specifici come `documentazione.pdf`, per il quale è stata implementata una **gestione manuale** all'interno del metodo `do_GET`. In caso di esistenza del file, viene restituito con header esplicito `Content-Type: application/pdf`, assicurando che il browser interpreti correttamente il file come un documento PDF, e non come testo generico.

Allo stesso modo, elementi grafici come il file `logo.png`, incluso nella homepage `index.html`, vengono serviti con header `image/png`, garantendo una corretta visualizzazione del logo nel browser.

3.5 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 sarè possibile visualizzare su console informazioni riguardo le richieste rievute 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 risora.

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 reindirazzamento 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 gia a disposizione del client e immutata dall'ultima trasmissione. Non risulta quindi necessario ritrasmetterla.