Relazione Tecnica: Web Server Minimo in Python

Lucas Prati

5 giugno 2025

Indice

1 Introduzione **2**

2 Struttura del codice **3**

3 Gestione degli errori e sicurezza **6**

4 File statici serviti **7**

5 Verifica e test **8**
5.1 Test via browser 8
5.2 Test via `curl` 9
5.3 Test automatizzato con `testServer.py` 9

6 Conclusioni **11**

Capitolo 1

Introduzione

In questa relazione viene descritto lo sviluppo di un Web Server minimale in Python che utilizza socket a basso livello per servire pagine web statiche. Vengono illustrate le scelte architetturali, la struttura del codice, le misure di sicurezza adottate, i file statici serviti e le modalità di verifica tramite browser e curl, incluso uno script di test automatizzato.

Capitolo 2

Struttura del codice

Il file principale del progetto è `server.py`, che include le seguenti fasi:

- **Creazione del socket**
  Viene creato un socket TCP (IPv4) con:

```
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
serverSocket.bind((HOST, PORT))
serverSocket.listen(5)
```

  L'opzione `SO_REUSEADDR` consente di riutilizzare la porta anche se è in `TIME_WAIT`. Si effettua il binding a `localhost:8080` e si imposta un backlog pari a 5.

- **Accept e threading**
  In un ciclo `while True`, il server esegue `accept()`, che rimane in attesa di connessioni. Quando un client si connette, `accept()` restituisce un nuovo socket (`connectionSocket`) e l'indirizzo del client. Per ogni connessione, si crea un nuovo thread:

```
thread = threading.Thread(
    target=handle_client,
    args=(connectionSocket, client_address)
)
thread.daemon = True
thread.start()
```

  Questo permette di gestire più client contemporaneamente senza bloccare il thread principale.

- **Parsing delle richieste HTTP**
  Nella funzione `handle_client(connectionSocket, client_address)`:
  - Si legge fino a 4096 byte tramite `connectionSocket.recv(4096)`.
  - Si decodifica in UTF-8 e si estrae la prima riga con `splitlines()[0]`, che ha la forma `"GET /nomefile HTTP/1.1"`.
  - Se il metodo non è `GET`, il server risponde con:

```
HTTP/1.1 501 Not Implemented
Content-Type: text/html
Content-Length: <lunghezza>
```

```
<html><body><h1>501 Not Implemented</h1></body></html>
```

e chiude la connessione.

- Se il metodo è `GET`, si determina il nome del file richiesto:
  - Se il percorso è `/` o `/index.html`, si serve `index.html`.
  - Altrimenti si rimuove lo slash iniziale con `filename = path.lstrip('/')`.

- **Determinazione e sicurezza del file richiesto**
  Il percorso del file è costruito concatenando la cartella `WWW_DIR` ("www") e il nome del file:

```
resource_path = os.path.join(WWW_DIR, filename)
abs_resource = os.path.abspath(resource_path)
abs_www_dir  = os.path.abspath(WWW_DIR)
if not abs_resource.startswith(abs_www_dir + os.sep):
    raise FileNotFoundError
```

  In questo modo si evita la directory traversal: se `abs_resource` non si trova in `WWW_DIR`, si solleva `FileNotFoundError` e si risponde con 404.

- **Lettura e invio del file**
  Se il file viene trovato, viene aperto in modalità binaria:

```
with open(abs_resource, 'rb') as f:
    content = f.read()
```

  Si determina il `Content-Type` in base all'estensione, tramite un dizionario `MIME_TYPES`, ad esempio:

```
MIME_TYPES = {
    '.html': 'text/html',
    '.css':  'text/css',
    '.jpg':  'image/jpeg',
    '.jpeg': 'image/jpeg',
    '.png':  'image/png',
    '.gif':  'image/gif'
}
content_type = MIME_TYPES.get(ext.lower(), 'application/octet
   -stream')
```

  L'header di risposta viene costruito come:

```
HTTP/1.1 200 OK
Content-Type: <content_type>
Content-Length: <len(content)>
```

  e inviato insieme al corpo binario con:

```
connectionSocket.sendall(header.encode())
connectionSocket.sendall(content)
connectionSocket.close()
```

- **Gestione del 404**
  Se il file non esiste o non è valido, si cattura il `FileNotFoundError`:

```
<html>
  <head>
    <title>404 Not Found</title>
  </head>
  <body>
    <h1>404 Not Found</h1>
    <p>Il file '<span>\texttt{filename}</span>' non &egrave;
       stato trovato.</p>
  </body>
</html>
```

  L'header inviato è:

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: <len(body)>
```

  Dopo aver inviato header e body, si chiude la connessione.

- **Logging delle richieste**
  Ogni volta che si serve un file (200), si restituisce un 404 o un 501, viene chiamata:

```
log_request(client_address, request_line, response_code,
   resource)
```

  Questa funzione stampa su console:

  `[REQUEST] <IP>:<porta> "<request_line>" -> <codice> (file:  <resource>)`

  consentendo di monitorare in tempo reale ogni richiesta ricevuta.

Capitolo 3

Gestione degli errori e sicurezza

- **Directory traversal**
  Utilizzando `os.path.abspath()` e verificando che il percorso assoluto di ogni risorsa inizi con il percorso assoluto di `WWW_DIR`, si prevengono tentativi di accedere a file al di fuori della directory `www`.
- **Metodo non supportato**
  Se un client invia un metodo diverso da `GET`, il server risponde con `501 Not Implemented` e chiude la connessione.
- **Eccezioni generiche**
  Tutti i blocchi potenzialmente critici (lettura socket, apertura file, invio dati) sono racchiusi in blocchi `try/except`. In caso di eccezioni non previste, il server chiude il socket, evitando crash imprevisti.

Capitolo 4

File statici serviti

La cartella `www` contiene i seguenti file, aggiornati dopo le ultime modifiche:

- **index.html** Pagina di benvenuto, con un breve testo, una lista puntata e una barra di navigazione verso le altre pagine.
- **pagina1.html** Contiene un titolo, una barra di navigazione verso le altre pagine e l'immagine `animal.jpg`
- **pagina2.html** Contiene un titolo, una barra di navigazione verso le altre pagine e una GIF animata `dance.gif`
- **pagina3.html** Contiene un titolo, una barra di navigazione verso le altre pagine e un logo `unibo.png`
- **styles.css** Definisce gli stili base, l'animazione CSS e il layout responsive (modifica **solo sfondo e margini** per schermi ≤ 600 px). In questo modo, passando a schermi stretti, si nota subito il cambio di **sfondo** e l'aumento dei **margini laterali** e inferiori.
- **Immagini** (cartella `www/images/`):
  - `animal.jpg` (JPEG)
  - `dance.gif` (GIF animata)
  - `unibo.png` (PNG del logo UNIBO)

  Tutti i file sono serviti con il MIME type corretto dedotto dall'estensione.

Capitolo 5

Verifica e test

Per avviare il server, dalla cartella principale (dove si trova `server.py`), eseguire:

```
python3 server.py
```

Per cambiare porta:

```
python3 server.py 8080
```

Sul terminale compariranno:

```
Web Server avviato: localhost:8080
Document root: ./www/
Premi CTRL-C per terminare
```

5.1 Test via browser

- **Home**
  Collegarsi a `http://localhost:8080/` → si carica `index.html` (pagina di benvenuto).
- **Pagina 1**
  Cliccare su "Pagina 1" → `pagina1.html` (contiene `animal.jpg`).
- **Pagina 2**
  Cliccare su "Pagina 2" → `pagina2.html` (contiene `dance.gif`).
- **Pagina 3**
  Cliccare su "Pagina 3" → `pagina3.html` (contiene `unibo.png`).
- **404 Not Found**
  Provare a inserire manualmente un percorso inesistente, `http://localhost:8080/nonEsiste.html` → appare la pagina 404 personalizzata.
- **Layout responsive**
  Su desktop (>600 px) si vede **sfondo chiaro** e `nav` in orizzontale. Riducendo la finestra sotto i 600 px, si nota **subito il cambio di sfondo**, **margini laterali** più ampi e `nav` in colonna verticale con margini tra i link.
- **Animazione CSS**
  In tutte le pagine, il tag `<h1>` presenta l'animazione definita con `@keyframes pulsante`, che fa "pulsare" il colore dal grigio al rosso ciclicamente.

5.2 Test via `curl`

- **Pagine HTML e CSS**:

```
curl -i http://localhost:8080/index.html
curl -i http://localhost:8080/pagina1.html
curl -i http://localhost:8080/pagina2.html
curl -i http://localhost:8080/pagina3.html
curl -i http://localhost:8080/styles.css
```

  Tutti devono restituire `HTTP/1.1 200 OK` con il corretto `Content-Type` (`text/html` o `text/css`).

- **Scaricare le immagini** (senza header):

```
curl http://localhost:8080/images/animal.jpg \
     --output prova_animal.jpg

curl http://localhost:8080/images/dance.gif \
     --output prova_dance.gif

curl http://localhost:8080/images/unibo.png \
     --output prova_logo.png
```

- **404 Not Found**:

```
curl -i http://localhost:8080/nonEsiste.html
```

  Deve restituire `HTTP/1.1 404 Not Found` con il body di errore personalizzato.

- **501 Not Implemented**:

```
curl -X POST -i http://localhost:8080/index.html
```

  Deve restituire `HTTP/1.1 501 Not Implemented`.

5.3 Test automatizzato con `testServer.py`

Nel progetto è incluso lo script `testServer.py`, che verifica automaticamente i seguenti percorsi:

```
"/"
"/index.html"
"/pagina1.html"
"/pagina2.html"
"/pagina3.html"
"/styles.css"
"/images/animal.jpg"
"/images/dance.gif"
"/images/unibo.png"
"/nonEsiste.html"
```

Lo script si connette a `localhost:8080`, invia richieste HTTP GET di tipo:

```
GET <path> HTTP/1.1
Host: localhost:8080
```

e stampa la prima riga della risposta, ad esempio:

```
/                   → HTTP/1.1 200 OK
/index.html         → HTTP/1.1 200 OK
/pagina1.html       → HTTP/1.1 200 OK
/pagina2.html       → HTTP/1.1 200 OK
/pagina3.html       → HTTP/1.1 200 OK
/styles.css         → HTTP/1.1 200 OK
/images/animal.jpg  → HTTP/1.1 200 OK
/images/dance.gif   → HTTP/1.1 200 OK
/images/unibo.png   → HTTP/1.1 200 OK
/nonEsiste.html     → HTTP/1.1 404 Not Found
```

In questo modo si verifica rapidamente che tutte le risorse esistenti restituiscano 200 e quelle mancanti restituiscano 404.

Capitolo 6

Conclusioni

Questo progetto permette di comprendere i concetti basilari di socket programming con Python e il meccanismo di richiesta/risposta HTTP a basso livello. Sono stati soddisfatti tutti i requisiti minimi:

- Il server risponde su `localhost:8080`.
- Servono quattro pagine HTML statiche (`index.html`, `pagina1.html`, `pagina2.html`, `pagina3.html`) e un file CSS (`styles.css`) che include animazioni CSS e un semplice layout responsive.
- Sono gestite le richieste:
  - `GET` → `200 OK`
  - Risorsa inesistente → `404 Not Found`
  - Metodo diverso da GET → `501 Not Implemented`
- È implementato il logging su console per ogni richiesta, con stampa di IP, request line, codice di risposta e risorsa.
- Lo script di test `testServer.py` verifica automaticamente la corretta gestione di tutte le risorse principali.