

Elaborato per il corso  
”Programmazione di Reti”

Alessandro Rebosio

30 settembre 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Architettura del progetto</b>	<b>3</b>
2.1	Descrizione dell'architettura . . . . .	3
<b>3</b>	<b>Implementazione del web server</b>	<b>4</b>
3.1	Struttura generale . . . . .	4
3.2	Gestione delle richieste . . . . .	4
3.3	Gestione MIME types e logging . . . . .	5
3.4	Sicurezza base . . . . .	5
<b>4</b>	<b>Sito web statico</b>	<b>6</b>
4.1	Pagine HTML . . . . .	6
4.2	Estendibilità . . . . .	6
<b>5</b>	<b>Test e risultati</b>	<b>7</b>
5.1	Test funzionali . . . . .	7
5.1.1	Richieste valide (200 OK) . . . . .	7
5.1.2	Risorse non trovate (404 Not Found) . . . . .	8
5.1.3	Metodo non supportato (501 Not Implemented) . . . . .	8
5.1.4	Test sui MIME . . . . .	9
5.1.5	Conclusioni dei test . . . . .	9
<b>6</b>	<b>Conclusioni</b>	<b>10</b>
6.1	Sviluppi futuri . . . . .	10
<b>A</b>	<b>Guida utente</b>	<b>11</b>
A.1	Clonazione del repository . . . . .	11
A.2	Esecuzione del server . . . . .	11
A.3	Accesso alle pagine . . . . .	11

# Capitolo 1

## Introduzione

L'obiettivo di questo progetto è la realizzazione di un **server HTTP** sviluppato in **Python**, capace di gestire richieste di tipo **GET** sulla **porta 8080** di **localhost** e di servire contenuti statici in formato **HTML** e **CSS**.

Il progetto intende fornire una comprensione pratica del **protocollo HTTP**, dell'utilizzo dei **socket** per la comunicazione di rete e del funzionamento di base di un **web server**.

### Funzionalità principali

- Risposta con codice di stato 200 per risorse esistenti;
- Gestione dell'errore 404 per risorse non trovate;
- Riconoscimento e gestione dei MIME types per i file serviti;
- Logging delle richieste ricevute;
- Pubblicazione di un sito web statico composto da tre pagine HTML responsive.

# Capitolo 2

## Architettura del progetto

L'architettura del progetto si basa su una **suddivisione** tra la logica applicativa del server e i contenuti statici da servire al client.

### 2.1 Descrizione dell'architettura

- **src/server.py**: implementa il web server HTTP, ascolta sulla porta 8080 e serve file statici presenti nella cartella **www/**.
- **www/**: contiene il sito web statico, costituito da tre pagine HTML e un file CSS condiviso. Le pagine includono layout responsive e contenuti base.

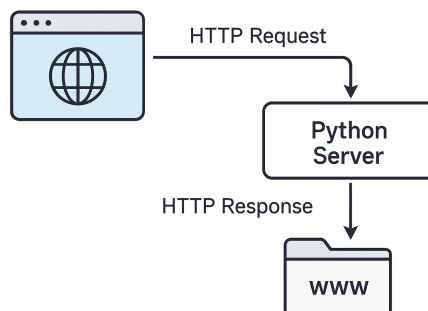


Figura 2.1: Schema dell'architettura client-server

In Figura 2.1 viene rappresentata l'interazione tra browser, server e file system. Quando un client invia una richiesta HTTP, il server interpreta la richiesta, individua il file richiesto nella directory **www/** e restituisce il contenuto.

# Capitolo 3

## Implementazione del web server

L'implementazione del server HTTP è interamente in `Python` utilizzando il modulo `socket` per la comunicazione e il modulo `threading` per la gestione concorrente delle connessioni.

### 3.1 Struttura generale

- **HTTPServer**: classe usata per stabilire una o più connessioni attraverso la creazione di un `socket`, che viene poi associato alla porta 8080 per accettare le connessioni in ingresso.
- **HTTPRequestHandler**: classe per il parsing delle richieste con il successivo invio di risposte.
- **MyServer**: sottoclasse per semplificare la logica, nella quale viene implementato il metodo `do_GET` per servire i file richiesti dai client.

### 3.2 Gestione delle richieste

Quando un client tenta di stabilire una connessione, il server la accetta e crea un nuovo `thread` dedicato alla sua gestione, consentendo l'elaborazione concorrente di più richieste.

La richiesta viene quindi letta e, dalla prima riga, vengono estratti il `metodo`, il `percorso` (`path`) e la `versione HTTP`.

Successivamente, se la risorsa richiesta è presente nella directory `www/`, essa viene caricata e restituita al client con una risposta contenente il codice (200 OK). In caso contrario, viene restituita una risposta di errore con codice.

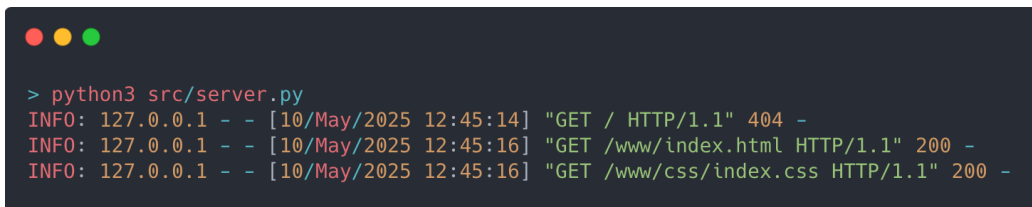
### 3.3 Gestione MIME types e logging

Il server usa la libreria `mimetypes` per determinare il tipo di contenuto da servire in base all'estensione del file.

Ogni richiesta viene registrata in console Figura 3.1 con il seguente formato:

```
ip - - [data] "metodo risorsa versione" codice -
```

#### Esempio di log delle richieste



```
> python3 src/server.py
INFO: 127.0.0.1 - - [10/May/2025 12:45:14] "GET / HTTP/1.1" 404 -
INFO: 127.0.0.1 - - [10/May/2025 12:45:16] "GET /www/index.html HTTP/1.1" 200 -
INFO: 127.0.0.1 - - [10/May/2025 12:45:16] "GET /www/css/index.css HTTP/1.1" 200 -
```

Figura 3.1: Esempio di log delle richieste stampate in console

### 3.4 Sicurezza base

Per evitare accessi non autorizzati a file esterni alla cartella `www/`, il path richiesto viene sanificato rimuovendo sequenze potenzialmente pericolose come `../`.

#### Sicurezza del path:



```
self.path.lstrip("/").replace("../", "").replace("../\\", "")
```

Figura 3.2: Codice per path sicura

# Capitolo 4

## Sito web statico

Il sito web servito dal web server, al momento, è composto da tre pagine HTML statiche, con layout responsive e stile definito tramite file CSS dedicati. L'obiettivo è fornire un'interfaccia semplice per dimostrare il corretto caricamento dei contenuti da parte del server.

### 4.1 Pagine HTML

- **index.html** - Homepage che presenta una navigazione verso le altre sezioni del sito.
- **login.html** — Simula una pagina di login: nel caso venga inviata una richiesta POST, il server restituirà un errore 501 (metodo non implementato).
- **contact.html** — Pagina di contatto, con pulsanti per tornare alla home o alla pagina di login.

*Tutte le pagine fanno uso del framework **Bootstrap 5** per garantire un design responsive e compatibile con dispositivi di varie dimensioni.*

### 4.2 Estendibilità

Il sistema è progettato per essere facilmente estendibile: per aggiungere una nuova pagina è sufficiente **creare** un file HTML nella cartella **www/** e assicurarsi che il **collegamento** nel sito punti al percorso corretto.

*Non è necessario modificare il codice del web server: qualsiasi file esistente sarà automaticamente servito, a condizione che venga effettuata una richiesta HTTP valida.*

# Capitolo 5

## Test e risultati

Al fine di verificare il corretto funzionamento del web server, sono stati condotti diversi test utilizzando lo strumento da linea di comando `curl`, utile per simulare richieste HTTP verso il server.

### 5.1 Test funzionali

#### 5.1.1 Richieste valide (200 OK)

Sono state effettuate richieste HTTP di tipo `GET` verso le pagine HTML presenti nella cartella `www/`.

```
> curl -I http://localhost:8080/www/index.html
```

**Risposta:**

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 872
```

#### Verifica Browser

La stessa risorsa è accessibile anche tramite browser web all'URL:

- `http://localhost:8080/www/index.html`



### 5.1.2 Risorse non trovate (404 Not Found)

È stata verificata la gestione degli errori accedendo a una risorsa non presente nella cartella `www/`. Il server ha risposto correttamente con un errore **404 Not Found**.

```
> curl -I http://localhost:8080/www/missing.html
```

**Risposta:**

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 1245
```

#### Verifica Browser

La stessa risorsa è accessibile anche tramite browser web all'URL:

- `http://localhost:8080/`

### 5.1.3 Metodo non supportato (501 Not Implemented)

È stata testata la risposta del server a una richiesta POST:

```
> curl -I -X POST http://localhost:8080/
```

**Risposta:**

```
HTTP/1.1 501 Not Implemented
Content-Type: text/html
Content-Length: 1263
```

#### Verifica Browser

La stessa risorsa è accessibile anche tramite browser web all'URL, quando si tenta di inviare il form di login:

- `http://localhost:8080/www/login.html`

*Questo risultato è coerente con le specifiche del progetto, che prevedono l'implementazione solo del metodo GET per la gestione di richieste HTTP.*

### 5.1.4 Test sui MIME

È stato verificato il corretto riconoscimento dei tipi MIME da parte del server. I test hanno confermato che il server identifica accuratamente il tipo di contenuto e restituisce l'appropriato **Content-Type** nell'header della risposta.

```
> curl -I http://localhost:8080/www/img/arc.png
```

La risposta del server mostra il **corretto** tipo MIME per un'immagine PNG:

```
HTTP/1.1 200 OK
Content-Type: image/png
Content-Length: 1171993
```

### Verifica Browser

La stessa risorsa è accessibile anche tramite browser web all'URL:

- <http://localhost:8080/www/img/arc.png>

con corretta visualizzazione dell'immagine.

### 5.1.5 Conclusioni dei test

Tutti i test funzionali hanno confermato la robustezza del server nell'ambito dei requisiti minimi. Le estensioni implementate (gestione MIME, logging) hanno migliorato l'usabilità e la compatibilità con i browser moderni. La struttura modulare facilita inoltre futuri ampliamenti.

# Capitolo 6

## Conclusioni

Il progetto ha raggiunto tutti gli obiettivi prefissati, realizzando un web server HTTP con le funzionalità essenziali, utilizzando un linguaggio di programmazione ad alto livello come Python. Attraverso questo sviluppo sono stati approfonditi diversi aspetti fondamentali:

- Il funzionamento del protocollo HTTP/1.1
- La gestione delle connessioni tramite socket
- L'elaborazione delle richieste e la costruzione delle risposte
- La gestione concorrente delle connessioni
- La sicurezza di base nell'accesso alle risorse

Il server sviluppato, pur nella sua semplicità, dimostra di essere in grado di gestire correttamente le richieste GET per file statici, con appropriate risposte di errore per risorse non trovate o metodi non implementati. L'integrazione con un sito web reale, seppur minimale, ha permesso di verificare il corretto funzionamento in uno scenario pratico.

### 6.1 Sviluppi futuri

Possibili estensioni e miglioramenti includono:

- Implementazione del metodo POST per form interattivi
- Supporto per l'upload di file
- Aggiunta di un sistema di caching

# Appendice A

## Guida utente

### A.1 Clonazione del repository

Clonare il progetto da GitHub e accedere alla cartella:

```
> git clone https://github.com/alessandrorebosio/NET25-httpServer.git  
> cd nNET25-httpServer  
sudo tlmgr update --self
```

### A.2 Esecuzione del server

Il server può essere avviato eseguendo lo script principale:

```
> python src/server.py
```

*Il server sarà attivo su `http://localhost:8080`.*

### A.3 Accesso alle pagine

Dal browser è possibile accedere alle pagine, presenti nella directory `www/`:

- `http://localhost:8080/www/index.html`
- `http://localhost:8080/www/login.html`
- `http://localhost:8080/www/contact.html`

*Se non presenti, verrà restituito l'errore `404 - Not Found`.*