Elaborato per il corso
*Programmazione di Reti*

Grazia Bochdanovits de Kavna

29 agosto 2025

# Indice

# Capitolo 1

# Introduzione

Il presente progetto ha l'obiettivo di realizzare un **web server HTTP** in **Python**, realizzato utilizzando socket per gestire le connessioni di rete. Il server è progettato per rispondere sulla **porta 8080** di `localhost` e servire contenuti statici dalla directory `www/`.

Il lavoro implementa tutti i requisiti minimi richiesti dalla traccia: gestione delle richieste `GET` con risposta 200, servizio di almeno tre pagine HTML statiche (`index.html`, `login.html`, `contact.html`), e corretta gestione degli errori 404 per file inesistenti.

Sono state inoltre implementate le estensioni opzionali previste, tra cui il supporto per *MIME types* (HTML, CSS), il logging delle richieste con *timestamp*, e un layout responsive con animazioni CSS.

Il server utilizza **multithreading** per gestire connessioni concorrenti e include misure di sicurezza per prevenire attacchi di directory traversal. Il progetto comprende anche pagine di errore personalizzate (`404.html`, `501.html`) e un design moderno.

# Capitolo 2

# Architettura del progetto

L'architettura del progetto si basa su una chiara separazione tra la logica applicativa del server e i contenuti statici destinati al client. Il sistema è organizzato nei seguenti componenti principali:

- **server.py**: implementa il web server HTTP multithread, in ascolto sulla porta 8080, responsabile della gestione delle connessioni e del servizio dei file statici.
- **www/**: directory contenente il sito web statico, composto da cinque pagine HTML e un file CSS condiviso, con layout responsive e design moderno.

# Capitolo 3

# Implementazione del web server

## 3.1 Struttura generale

L'implementazione del web server è stata realizzata attraverso una classe Python `HTTPServer` che incapsula tutte le funzionalità necessarie per servire contenuti statici via HTTP. La struttura del server segue un approccio object-oriented che favorisce modularità e manutenibilità del codice.

### 3.1.1 Architettura della classe `HTTPServer`

La classe principale `HTTPServer` è organizzata in componenti specifici, ognuno implementato attraverso metodi dedicati:

- **Configurazione iniziale**: I metodi `__init__` e `_initialize_mime_types` gestiscono l'inizializzazione del server e la configurazione dei tipi MIME supportati.
- **Gestione socket**: I metodi `_setup_socket`, `_bind_and_listen` e `_cleanup` si occupano della configurazione e gestione del socket di rete durante tutto il ciclo di vita del server.
- **Elaborazione richieste**: I metodi `handle_client`, `_process_request` e `handle_get_request` implementano il parsing e il processamento delle richieste HTTP.
- **Generazione risposte**: I metodi `serve_file`, `send_error`, `_build_response_headers` e `_send_response` costruiscono e inviano le risposte HTTP appropriate.
- **Gestione concorrenza**: I metodi `_handle_client_in_thread` e il main loop `_run_server_loop` gestiscono le connessioni concorrenti attraverso thread separati.

- **Controlli sicurezza**: I metodi `_resolve_file_path` e `_is_safe_path` implementano i controlli di sicurezza per prevenire accessi non autorizzati al filesystem.
- **Utility**: Metodi di supporto come `log_request`, `_read_file_content`, `_get_mime_type` forniscono funzionalità ausiliarie.

### 3.1.2 Ciclo di vita del Server

Il server opera secondo un ciclo di vita ben definito:

1. **Inizializzazione**: Configurazione parametri e mappatura MIME types attraverso `__init__` e `_initialize_mime_types`.
2. **Startup**: Creazione socket e binding all'indirizzo specificato tramite `_setup_socket` e `_bind_and_listen`.
3. **Main loop**: Accettazione connessioni client nel metodo `_run_server_loop`, con delega a `_handle_client_in_thread` per la gestione in thread separati.
4. **Shutdown**: Chiusura ordinata delle risorse attraverso `stop` e `_cleanup`.

## 3.2 Gestione delle richieste

### 3.2.1 Validazione dei metodi HTTP

Il server implementa una whitelist approach, accettando esclusivamente richieste GET per garantire sicurezza e semplicità:

```
if method != 'GET':
    self.send_error(client_socket, 405, "Method Not Allowed")
    return
```

### 3.2.2 Risoluzione dei path

La risoluzione dei path URL avviene attraverso un processo di parsing e normalizzazione che include controlli di sicurezza per prevenire directory traversal attacks:

```
def _resolve_file_path(self, url_path: str) -> Optional[str]:
    parsed_path = urlparse(url_path)
    path = unquote(parsed_path.path)
    # ... normalizzazione e controlli di sicurezza
```

### 3.2.3 Threading e concorrenza

Ogni richiesta client viene gestita in un thread separato, permettendo al server di servire multiple connessioni simultaneamente:

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

## 3.3 Gestione MIME types e logging

#### Headers HTTP

La generazione degli headers HTTP segue lo standard HTTP/1.1, includendo tutti i campi necessari per una corretta comunicazione client-server:

```
def _build_response_headers(self, status_code: int, status_text: str,
                            content_type: str, content_length: int) ->
    return [
        f"HTTP/1.1 {status_code} {status_text}",
        f"Content-Type: {content_type}",
        f"Content-Length: {content_length}",
        "Connection: close",
        "\r\n"
    ]
```

#### Sistema di logging

Il sistema di logging registra timestamp, indirizzo client e dettagli della richiesta, fornendo visibilità completa sull'attività del server:

```
def log_request(self, client_address: Tuple[str, int], request: str) -
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    request_line = request.split('\n')[0].strip() if request else "Inv
    print(f"[{timestamp}] {client_address[0]}:{client_address[1]} - {r
```

## 3.4 Sicurezza base

### 3.4.1 Prevenzione directory traversal

Il controllo di sicurezza verifica che il path richiesto sia contenuto entro la directory autorizzata, prevenendo accessi a file sensibili del sistema:

```
def _is_safe_path(self, path: str) -> bool:
    requested_path = os.path.abspath(path)
    return requested_path.startswith(self.www_dir)
```

### 3.4.2 Gestione degli errori HTTP

Il server implementa una gestione completa degli errori HTTP con codici di stato appropriati e pagine personalizzate per migliorare l'esperienza utente.

### 3.4.3 Resource cleanup

Il sistema di cleanup garantisce il corretto rilascio delle risorse di rete e la chiusura ordinata di tutte le connessioni.

# Capitolo 4

# Sito web statico

Il sito web è composto da cinque pagine HTML statiche, tutte stilizzate mediante un unico file CSS condiviso (`common.css`).

## 4.1 Pagine HTML

- **index.html**: Homepage del sito con navigazione principale
- **login.html**: Pagina di autenticazione con form per le credenziali
- **contact.html**: Pagina di contatto con informazioni e form
- **404.html**: Pagina di errore personalizzata per risorse non trovate
- **501.html**: Pagina per funzionalità non ancora implementate

## 4.2 Estendibilità

La struttura del sito è progettata per essere facilmente estendibile grazie all'utilizzo di un unico file CSS e una organizzazione chiara dei file, permettendo l'aggiunta di nuove pagine con minimo sforzo.

# Capitolo 5

# Conclusioni

Il server sviluppato in Python soddisfa pienamente i requisiti della traccia, rispondendo su `localhost:8080`, gestendo correttamente richieste GET e fornendo pagine di errore personalizzate. Sono state realizzate tutte le estensioni opzionali, includendo la gestione dei MIME type, il logging delle richieste e un sito statico con layout responsive. Il server rappresenta una soluzione ben strutturata che supera i requisiti minimi previsti.

## 5.1 Sviluppi Futuri

Il progetto, pur essendo completo e funzionale, può essere esteso con diverse funzionalità avanzate:

- **Supporto HTTPS**: Implementazione del protocollo TLS per comunicazioni cifrate.
- **Gestione metodi aggiuntivi**: Estensione del supporto ai metodi `POST` e `PUT`.
- **Sistema di caching**: Implementazione di meccanismi di caching per migliorare le performance.
- **Autenticazione avanzata**: Integrazione di un sistema di autenticazione con sessioni utente.
- **Supporto CGI**: Abilitazione dell'esecuzione di script CGI per contenuti dinamici.

# Appendice A

# Guida utente

## A.1 Installazione e Avvio

1. **Clonazione del repository**:
   `git clone https://github.com/this-Grace/ProgrammazioneReti-project.git`
2. **Accesso alla directory**:
   `cd ProgrammazioneReti-project`
3. **Avvio del server**:
   `python3 httpserver.py`
4. **Accesso al sito**:
   Aprire un browser e navigare su `http://localhost:8080`

## A.2 Accesso alle pagines

- **Homepage**: `http://localhost:8080/`
- **Pagina di login**: `http://localhost:8080/login.html`
- **Pagina di contatti**: `http://localhost:8080/contact.html`
- **Pagina 404**: `http://localhost:8080/404.html`
- **Pagina 501**: `http://localhost:8080/501.html`

### A.2.1 Note sull'accesso

- La root `/` restituisce automaticamente `index.html`.

- File non esistenti mostrano la pagina `404.html`.
- Il form di login reindirizza a `501.html`.

## A.3 Arresto del Server

- Premere `Ctrl+C` nel terminale.
- Attendere la chiusura delle connessioni.
- Verificare il messaggio: `Server stopped.`