

Università di Bologna - Campus di Cesena

Ingegneria e Scienze Informatiche (8615)

Documentazione

“Programmazione Di Reti”

WebServer – Traccia 2

Matricola: 0001077417 - Alex Mazzoni

Introduzione

Traccia progetto scelto (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.

Risoluzione del problema

Ho creato il web server semplice con il nome di 'server.py', si occupa di gestire le diverse richieste tra i diversi client, in sintesi, avvia un server http multithreaded che serve solo le medesime richieste GET, che il client ha richiesto, con i file contenuti nella directory './frontend/'.

Esecuzione del programma

Il programma scritto non presenta moduli esterni a quelli presenti nella libreria standard della versione di [Python corrente](#), per eseguirlo è necessario scaricare e installare Python.

Apri un terminale e esegui lo script con il comando:

```
python ./server.py
```

Comparirà a schermo la scritta `Starting server on http://localhost:8080`, con la pressione di `Ctrl+Click`, si effettuerà automaticamente una richiesta http sulla porta: 8080 al server.

Per interrompere il server, è necessaria la sola pressione di `Ctrl+C`.

Analisi del codice sorgente

Librerie

Vengono importati i moduli:

- `sys` → è un modulo che gestisce l'ambiente a Runtime, lo utilizzo per terminare l'esecuzione del processo, in modo pulito.
- `signal` → permette di gestire eventuali segnali, inviati dal processo o dal sistema operativo, lo utilizzo per intercettare la pressione di 'Ctrl+C'.
- `http.server` → è un server http semplice e basico incluso nella libreria standard di Python, implementa la funzionalità di web server.
- `socketserver` → fornisce un framework per la creazione di server, basati su socket Unix.



```
1 import sys, signal
2 import http.server
3 import socketserver
4
```

Configurazione globale del server

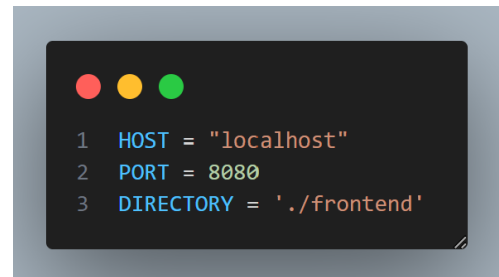
Inizializzo le variabili che utilizzerò per impostare il server con un determinato **indirizzo IP**, in questo caso ho scelto l'indirizzo ip localhost, corrisponde infatti all'indirizzo 127.0.0.1, viene automaticamente decifrato dal file hosts.

L'ho impostato in ascolto nella **porta 8080**, ma poteva essere anche scelta una qualsiasi porta, dato che i pacchetti vengono identificati da HOST+PORTA, è buona pratica però dare un numero di porta:

- Porta 80 per HTTP
- Porta 443 per HTTPS
- Porta 21 per FTP
- Porta 22 per SSH
- Porta 25 per SMTP

La porta 8080 è spesso utilizzata come alternativa alla porta 80 per il traffico HTTP.

La variabile **DIRECTORY** contiene il percorso assoluto della piccola porzione di File System che il server deve servire, in essa è contenuto un semplice sito HTML + JavaScript + CSS.



```
1 HOST = "localhost"
2 PORT = 8080
3 DIRECTORY = './frontend'
```

Creazione del gestore delle richieste

```
1 class SimpleHTTPHandler(http.server.SimpleHTTPRequestHandler):
2     def __init__(self, *args, **kwargs):
3         super().__init__(*args, directory=DIRECTORY, **kwargs)
4
5     def do_GET(self) -> None:
6         return super().do_GET()
```

Il modulo `http.server` fornisce due gestori (handler) per gestire le richieste http:

- `SimpleHTTPRequestHandler` -> Questa classe fornisce solo la gestione delle uniche richieste `'do_GET()'`, utili solo ed esclusivamente per servire file statici.
- `BaseHTTPRequestHandler` -> Questa classe fornisce oltre alla `'do_GET()'`, presente nella `Simple`, ma anche altre richieste come la `'do_POST()'` o la `'do_HEAD()'`.

Nel mio caso avevo bisogno di gestire solamente le richieste GET, per fornire file statici, per cui ho deciso di utilizzare la classe `SimpleHTTPRequestHandler`, nel mio caso ho creato la mia classe **`SimpleHTTPHandler`** che eredita la classe del modulo `'http.server.SimpleHTTPRequestHandler'`, poiché necessito di configurare una precisa directory da servire.

Ho modificato la funzione dunder `'__init__'`, che imposta la configurazione del gestore del server, e ho specificato la directory che ho dichiarato in **Configurazione globale del server**.

Creazione del server

```
1 server = socketserver.ThreadingTCPServer((HOST,PORT), SimpleHTTPHandler)
```

Crea un'istanza di un server TCP, con l'host e la porta specificati e una classe handler utile per la gestione delle richieste del server.

La Classe `ThreadingTCPServer` contenuta nel modulo `socketserver` è una classe che crea un nuovo thread per ogni richiesta, questo significa che il server è in grado di gestire più client simultaneamente.

Impostazione delle proprietà del server

```
1 server.daemon_threads = True
2 server.allow_reuse_address = True
```

Queste due impostazioni consentono di:

- **`daemon_threads`**: imposta i thread del server come daemon threads, essi infatti non impediscono al programma principale di uscire se tutti gli altri thread non daemon sono terminati.
- **`Allow_reuse_address`**: consente di riutilizzare un indirizzo che è già stato utilizzato in precedenza, con la presenza di questa impostazione se il server è stato appena riavviato non comparirà un errore di `'Address already in use'`.

Gestione dei segnali per la corretta terminazione del server

```
1 def signal_handler(signal, frame):
2     print( 'Exiting http server (Ctrl+C pressed)')
3     try:
4         if( server ):
5             server.server_close()
6     finally:
7         sys.exit(0)
8
9 signal.signal(signal.SIGINT, signal_handler)
```

La funzione **signal_handler** con l'utilizzo di un try/finally, se esiste, termina il server e il processo dell'applicazione.

Il modulo **sys** poi si occuperà di richiamare la funzione quando rileva il segnale '**signal.SIGINT**', che corrisponde alla pressione di '**Ctrl+C**'.

Avvio del server

```
1 try:
2     while True:
3         print("Starting server on http://{}:{}".format(HOST, PORT))
4         server.serve_forever()
5 except KeyboardInterrupt:
6     pass
7
8 server.server_close()
```

Avvio il server e lo faccio rimanere in esecuzione utilizzando un ciclo infinito, gestisco la chiusura con l'istruzione try/catch, appena viene segnalato l'evento **KeyboardInterrupt** interrompe il ciclo infinito.

È comunque necessario terminare il server, anche, nei casi dove lo script esce dal '**while true**' che tiene attivo il server, ma il segnale '**signal_handler**' non viene innescato.