Web Server

Overview

basic multi threaded HTTP server using Python's socket and threading libraries. It listens for incoming connections on port 6789, serves static files, and handles HTTP GET requests. If a requested file exists, it sends the file with appropriate headers. If not, it responds with a 404 Not Found error and displays a custom 404.html page.

Project Structure:

```
├── server.py          # Main server script
├── config.json        # Server configuration (port, default page, etc.)
├── HelloWorld.html    # Default landing page
├── 404.html           # Custom 404 error page
├── status.html        # Template for server status
```

How It Works

1.Server Setup

The server creates a socket with AF_INET and SOCK_STREAM (TCP).

serverSocket = socket(AF_INET, SOCK_STREAM)

Binds to all available interfaces on port 6789.

serverSocket.bind(('', serverPort))

Listens for up to 5 concurrent connections.

serverSocket.listen(5)

Request Handling

Only **GET** requests are supported.

If the path is `/`, it serves the default page

If the path is `/status`, it displays the server uptime dynamically.

Other paths attempt to serve static files.

If the file is not found, it returns a custom 404 page.

Steps to Run the Server:

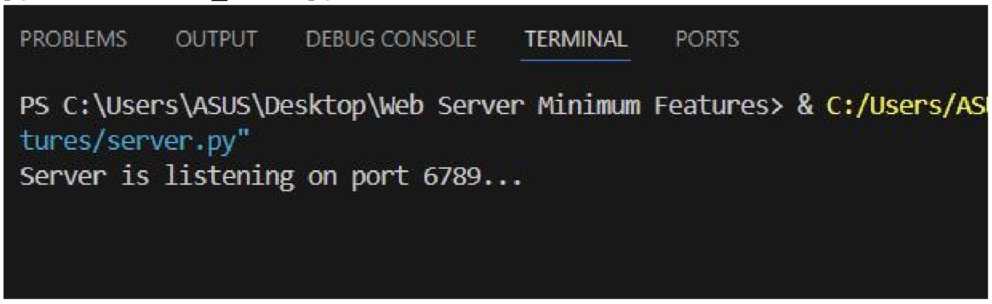
Steps to Run

1.Open terminal (or command prompt).

2.Navigate to the folder containing the server code.

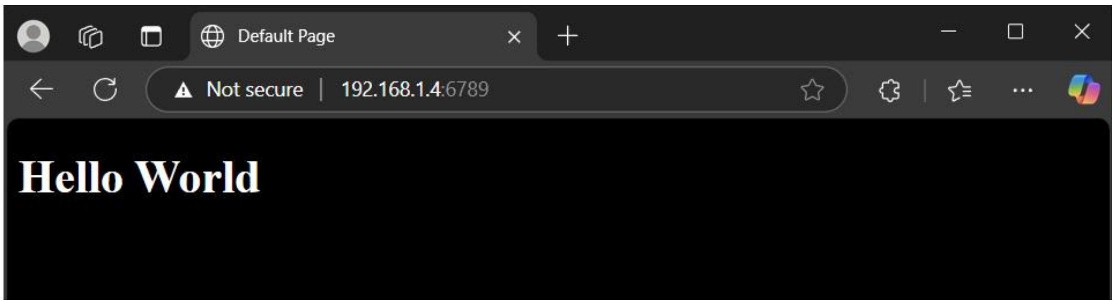
3.Run:

```
python server_file.py
```



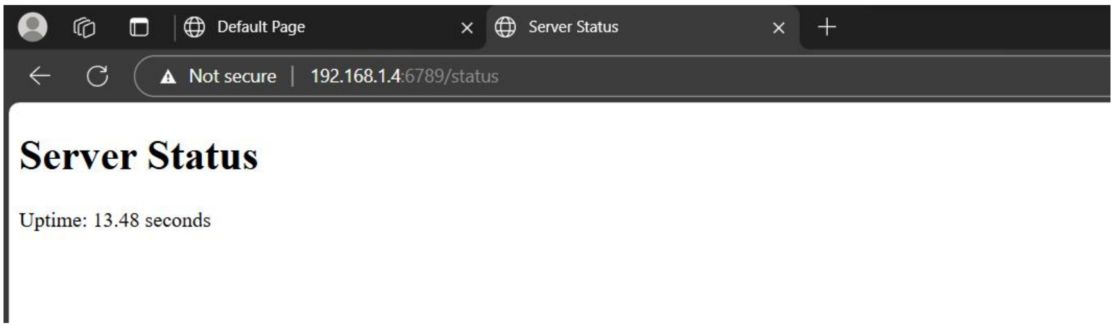
```
PS C:\Users\ASUS\Desktop\Web Server Minimum Features> & C:/Users/AS
tures/server.py"
Server is listening on port 6789...
```

4.   Visit http://Your_ip:6789 in your browser.



Hello World

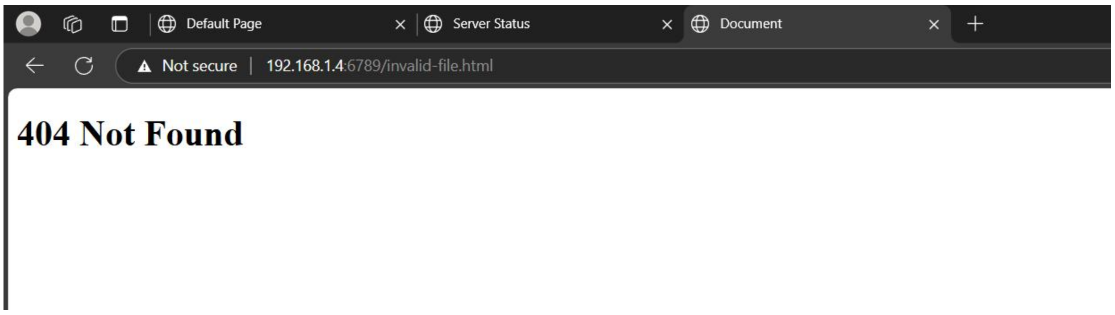
5.   http://ServerIP:6789/status



Server Status

Uptime: 13.48 seconds

6.http://ServerIP:6789/invalid-file.html


Default Page
Server Status
Document
Not secure | 192.168.1.4:6789/invalid-file.html

404 Not Found

What Does This Server Do?

This is a basic multithreaded HTTP server in Python. It:

Accepts multiple client connections.

Serves static HTML files from the local directory.

Displays uptime on a `/status` route.(The `/status` page is dynamic — it updates each time based on how long the server has been running.)

Responds with a `404` page if a file is not found.

Code Overview

*Imports

`socket`: For setting up TCP server sockets.

`threading`: For handling multiple connections at once.

`os`: To interact with the file system.

`mimetypes`: To determine file content types.

`json`: To load server configuration.

`time`: To calculate server uptime.

Main Components

```
with open('config.json', 'r') as config_file:
    config = json.load(config_file)
```

Reads server settings like port number and default page.

---

2. MIME Type Detection

`def get_mime_type(filename):`

Detects the file type based on the extension (`.html`, `.png`).

---

3. HTTP Response Formatting

`def format_response(status_code, content, content_type="text/html"):`

Creates a proper HTTP response with headers and body content.

---

4. Request Handling

`def handle_request(connection_socket, addr):`

Handles each client request:

- Parses the request line.
- Routes:
  - `/`: Loads default page
  - `/status`: Calculates and injects server uptime into HTML
  - other files: Attempts to load from disk
- Sends appropriate response:
  - 200 OK for valid file
  - 404 Not Found for missing file

---

5. `def start_server():`

Sets up and starts the server, using threads to handle multiple clients at once.

---

6. Entry Point

```
if __name__ == ″__main__″:
```

Ensures server starts when script is run directly.

Phase 2 – Possible Enhancements for the HTTP Server Project

Now that the basic HTTP server is up and running (handling static files and GET requests), here are several ideas to expand and enhance the project in **Phase 2**:

1. Add Support for POST Requests

Currently, the server only supports GET. In Phase 2, it can be extended to:

- Handle POST requests (e.g., submitting forms).
- Parse and process data sent in the request body.

Example use case: login form, contact form, etc.

2. Advanced Logging

- Save all incoming requests in an `access.log` file with details like time, client IP, and requested file.
- Save errors and exceptions in a separate `error.log` file.

3. Default index.html Support

If the user accesses a directory (e.g., `/`), the server should automatically look for and serve `index.html`, if it exists.

4. Configurable Server Settings

Instead of hardcoding things like port number or the root directory:

- Use a `config.json` file or
- Command-line arguments via `argparse` to configure the server behavior.

5. Basic Template Engine

Implement a simple HTML templating system:

- Allow dynamic content with placeholders like `{{username}}`.
- Replace them with actual values before sending the response.

6. Directory Listing

If a directory is requested and no `index.html` is found:

- Display a list of available files and folders in that directory as an HTML page.
- Useful for file browsers or local hosting.

7. HTTPS Support (Advanced)

- Use Python's `ssl` module to enable HTTPS.
- Create or use a self-signed certificate for local development.
- This improves security, especially for POST requests.

8. Performance Optimizations

- Implement caching for frequently accessed files.
- Use Gzip compression for large text files (HTML, CSS, JS).
- Replace basic threading with a **ThreadPool** to limit and manage threads more efficiently.

9. Add a Simple Admin Dashboard

Create a protected route (e.g., `/admin`) to show server stats:

- Total requests handled.
- Most requested files.
- Active threads or time of last request.

10. Add Basic REST API

- Add endpoints that return JSON instead of HTML.
- Example:

  ```
  bash
  CopyEdit
  GET /api/status → {"status": "OK"}
  ```

- This allows the server to act as a lightweight API backend.

Suggested Phase Breakdown:

Phase 1 Basic static file server with GET requests

Phase 2 Dynamic features like POST, templates, API, HTTPS, dashboard