HTTP: Dual-Stack TCP Server

Prepared by **Hamza Ghaffar (still in progress for more complex topics to learn)**

Header Types
1 Request Headers

Host: Specifies the domain name of the server (useful when multiple domains are hosted on a single IP). www.example.com

User-Agent: Contains information about the client software (browser type, version, operating system)

Accept: Specifies the media types that are acceptable for the response (e.g., HTML, JSON).
Example: **Accept: text/html,application/xhtml+xml**

Accept-Language: Indicates the preferred language for the response.

- Example: **Accept-Language: en-US,en;q=0.5**

Connection: Indicates whether the client wants to keep the connection alive after the current request.

- Example: **Connection: keep-alive**

Cache-Control: Directives for caching mechanisms in both requests and responses.

- Example: **Cache-Control: no-cache**

2 **Response Headers**

Status: Indicates the result of the request, including the status code and a description.

- Example: **HTTP/1.1 200 OK**

Content-Type: Indicates the media type of the resource being sent.

- Example: **Content-Type: application/json**

Content-Length: The size of the response body in bytes.

- Example: **Content-Length: 1234**

Server: Contains information about the server software.

- Example: **Server: Apache/2.4.1 (Unix)**

Cache-Control: Controls how the response should be cached.

- Example: **Cache-Control: no-store**

Set-Cookie: Used to send cookies from the server to the client.

- Example: **Set-Cookie: sessionId=abc123; HttpOnly**

1.1 Connection creation

```
dual_stack_socket.bind((host, port))
```

```
dual_stack_socket.listen()
```

```
"""the accept() function is a blocking call. This means that when
dual_stack_socket.accept() is reached, the server "pauses" here until a client
connects."""
client_socket, address = dual_stack_socket.accept()
```

1.2 Receiving the Request

```
request = client_socket.recv(1024).decode()

"""
# Request Line
GET /index.html HTTP/1.1
"""

# Headers - Key: Value

Host: example.com
User-Agent: Mozilla/5.0
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
"""

"""
# Body
{
    "key": "value"
}

"""
"""


lines = request.splitlines()
if len(lines) > 0:
    request_line = lines[0]
    method, path, _ = request_line.split()
```

2.1 Preparing the Response

```
"""
Prepare the HTTP response headers.
Status Line: This line indicates the HTTP version, the status code, and a status message

Response Headers: These headers provide additional information about the response, similar to request headers.

Blank-Space:

Response Body: This is the actual content being sent back to the client, such as HTML, JSON, images, etc.

"""

response = f"HTTP/1.1 200 OK\r\n" \
           f"Content-Type: application/json\r\n" \
           f"Content-Length: {len(response_body)}\r\n" \
           f"\r\n" \
           f"{response_body}"

# Send the response
client_socket.sendall(response.encode())
```

This guide will include a proper control flow to help you understand how everything fits together.

- **app/main.py**: This is the core file where your HTTP server code resides. It's the main entry point for your application's logic.
- **your_program.sh**: This is a shell script that provides a simple way to run your application locally.
- **.codecrafters/run.sh**: A specialized script used for the CodeCrafters platform, which provides an environment for running and testing code (such as server simulations).

- **Pipfile**: Defines your Python dependencies, listing the libraries required to run your application.
- **Pipfile.lock**: Locks the exact versions of dependencies used in the project to ensure consistency across different environments.

your_program.sh

```
#!/bin/bash

set -e  # Exit immediately if a command exits with a non-zero status

# Check if pipenv environment is set up and install dependencies if not
if [ ! -d ".venv" ]; then
  echo "Setting up the environment with pipenv..."
  pipenv install  # Install dependencies and create a virtual environment
fi

# Start the HTTP server within the pipenv environment
echo "Starting the HTTP server..."
```

```
exec pipenv run python3 -m app.main "$@"
```

Pipfile

The **Pipfile** is a configuration file used by Pipenv, a dependency manager for Python projects. This file defines the dependencies required for your project(While this file does not directly contain server code, it's crucial for ensuring you have the right packages installed to support your development.)

Pipfile.lock

Similar to **Pipfile**, this file locks the versions of the dependencies. It ensures that the same versions are used every time the project is set up, maintaining consistency across different environments. Like **Pipfile**, it's essential for the server but does not contain server code.

Improved Header Handling

```
# Step 1: Improved Header Handling
# Extracting All Headers: Implement a method to read and parse all request
headers to make decisions based on them.

headers = {}
for line in lines[1:]:
    if ': ' in line:
         key, value = line.split(': ', 1)
         headers[key] = value

# Display the headers in the console
 print("Extracted Headers:")
 for key, value in headers.items():
         print(f"{key}: {value}")
```

Read the Request Body

This is important because only **POST** requests will have a body that you want to read. If it's not a POST request, the server does not need to look for a body.
By default, in this approach, we cannot handle concurrent connections for more than 1 client. ~~its old fashion~~

```
# Read the Request Body
# Handling POST Requests: Implement logic to read the request body for POST
requests.
if method == "POST":
   content_length = int(headers.get('Content-Length', 0))
   body = request[request.index("\r\n\r\n") + 4:][:content_length]
   logging.info(f"Received body: {body}")  # Log the received body
```

Complete - example code

```
import socket
import logging
import json

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

def start_server(host='::', port=4221):
    try:
        dual_stack_socket = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
        dual_stack_socket.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_V6ONLY, 0)

        dual_stack_socket.bind((host, port))
        logging.info(f"Server listening on {host}:{port}...")

        dual_stack_socket.listen()

        while True:
            client_socket, address = dual_stack_socket.accept()
            logging.info(f"Connection from {address} has been established!")
            logging.info("Client socket info:")
            logging.info(f"Client Address: {address}")
            logging.info(f"Socket Family: {client_socket.family}")
            logging.info(f"Socket Type: {client_socket.type}")
            logging.info(f"Protocol: {client_socket.proto}")
            logging.info(f"File Descriptor: {client_socket.fileno()}")

            try:
                while True:
                    request = client_socket.recv(1024).decode()
                    if not request:
                        break

                    logging.info("Received request:")
                    logging.info(request)

                    lines = request.splitlines()
                    if len(lines) > 0:
                        request_line = lines[0]
                        method, path, _ = request_line.split()

                        # Step 1: Improved Header Handling
                        # Extracting All Headers: Implement a method to read and
parse all request headers to make decisions based on them.
                        headers = {}
                        for line in lines[1:]:
                            if ': ' in line:
                                key, value = line.split(': ', 1)
                                headers[key] = value

                        # Display the headers in the console
                        print("Extracted Headers:")
                        for key, value in headers.items():
                            print(f"{key}: {value}")

                        # Step 2: Read the Request Body
                        # Handling POST Requests: Implement logic to read the
request body for POST requests.
                        if method == "POST":
                            content_length = int(headers.get('Content-Length',
0))
                            body = request[request.index("\r\n\r\n") +
```

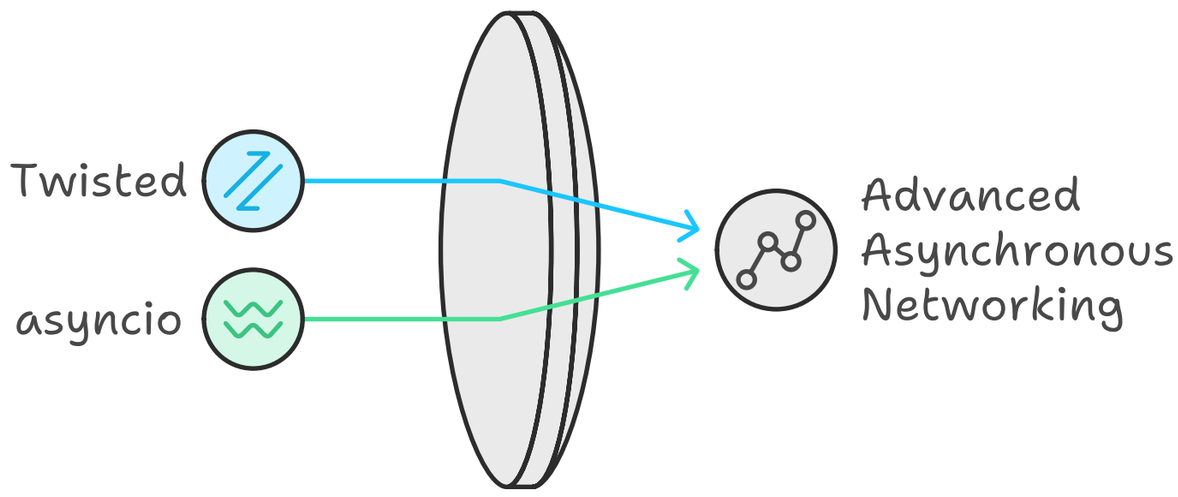
Now we can achieve this setup via asyncio

```
import asyncio
import logging
import json

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

async def handle_client(reader, writer):
    address = writer.get_extra_info('peername')
    logging.info(f"Connection from {address} has been established!")

    try:
        while True:
            data = await reader.read(1024)
            if not data:
                break  # Client closed the connection

            request = data.decode()
            logging.info("Received request:")
            logging.info(request)

            # Process the request
            lines = request.splitlines()
            if len(lines) > 0:
                request_line = lines[0]
                method, path, _ = request_line.split()

                # Step 1: Improved Header Handling
                headers = {}
                for line in lines[1:]:
                    if ': ' in line:
                        key, value = line.split(': ', 1)
                        headers[key] = value

                logging.info("Extracted Headers:")
                for key, value in headers.items():
                    logging.info(f"{key}: {value}")

                # Step 2: Read the Request Body
                if method == "POST":
                    content_length = int(headers.get('Content-Length', 0))
                    body = request[request.index("\r\n\r\n") +
4:][:content_length]
                    logging.info(f"Received body: {body}")

                # Prepare the response
                response_content = {
                    "message": "Hamza Ghaffar | Request received Successfully!",
                    "method": method,
                    "path": path
                }
                response_body = json.dumps(response_content)

                response = f"HTTP/1.1 200 OK\r\n" \
                           f"Content-Type: application/json\r\n" \
                           f"Content-Length: {len(response_body)}\r\n" \
                           f"\r\n" \
                           f"{response_body}"
```

Return a File -Example

```
import asyncio
import logging
import json
import os

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

# Use absolute path for the file
file_path = os.path.join(os.path.dirname(__file__), 'your_file.txt')  # Ensure
correct path

async def handle_client(reader, writer):
    address = writer.get_extra_info('peername')
    logging.info(f"Connection from {address} has been established!")

    try:
        while True:
            data = await reader.read(1024)
            if not data:
                break  # Client closed the connection

            request = data.decode()
            logging.info("Received request:")
            logging.info(request)

            # Process the request
            lines = request.splitlines()
            if len(lines) > 0:
                request_line = lines[0]
                method, path, _ = request_line.split()

                # Improved Header Handling
                headers = {}
                for line in lines[1:]:
                    if ': ' in line:
                        key, value = line.split(': ', 1)
                        headers[key] = value

                logging.info("Extracted Headers:")
                for key, value in headers.items():
                    logging.info(f"{key}: {value}")

                # Read the Request Body if POST
                if method == "POST":
                    content_length = int(headers.get('Content-Length', 0))
                    body = request[request.index("\r\n\r\n") +
4:][:content_length]
                    logging.info(f"Received body: {body}")

                # Prepare the response based on method and path
                if method == "GET" and path == "/your_file":  # Change this path
as needed
                    try:
                        with open(file_path, 'r') as file:
                            file_contents = file.read()

                        response_body = file_contents
                        response = f"HTTP/1.1 200 OK\r\n" \
                                   f"Content-Type: text/plain\r\n" \
                                   f"Content-Length: {len(response_body)}\r\n" \
```

You could use third-party libraries like Twisted or asyncio for more advanced asynchronous networking capabilities



Custom Loggers | Multiple Handlers | Logging Exceptions | Thread Safety

Grafana and Prometheus

General headers, entity headers, and custom headers that serve different purposes in the communication process.

Enhanced Logging: Implement structured logging using libraries like **structlog** to track requests and responses in production.

Security and Validation

Use a Reverse Proxy

Containerization