

Lecture 4: Create HTTP web server

This lecture presents how to build simple WebServer, that listens to port 8080 on address 127.0.0.1

```
1 # find process listening to port 8080
2 lsof -i :8080
```

You can use the `lsof` (List Open Files) command along with `grep` to find the process listening on port 8080 and then use the `kill` command to terminate it. Here's the command you can use in unix terminal:

```
1 lsof -i :8080 | awk '{print $2}' | grep -Eo '[0-9]+' | xargs 'kill'
```

Break down what this command does:

1. `lsof -i :8080`: Lists all open files (including network connections) and filters for those involving port 8080.
2. `awk '{print $2}'`: Extracts the second column, which contains the process IDs (PIDs) of the processes.
3. `xargs kill`: Takes the PIDs and sends a `kill` command to terminate the associated processes.

This command will find and terminate any process listening on port 8080.

1. Socket Listening (Listening to 127.0.0.1 Port 8080)

```
1 import socket
2
3 # Define the server's IP address and port
4 HOST = '127.0.0.1' # IP address to bind to (localhost)
5 PORT = 8080        # Port to listen on
6
7 # Create a socket that uses IPv4 and TCP
8 server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9
10 # Bind the socket to the address and port
11 server_socket.bind((HOST, PORT))
12
13 # Listen for incoming connections
14 server_socket.listen(5) # Backlog for multiple simultaneous connections
15 print(f"Server is listening on {HOST}:{PORT}")
16
17 client_socket, client_address = server_socket.accept()
18 print(client_address)
```

2. Single-Threaded Web Server (Receiving and Printing Request)

```
1 while True:
2     # Accept incoming client connections
3     client_socket, client_address = server_socket.accept()
4     print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
5
6     # Receive and print the client's request data
7     request_data = client_socket.recv(1024).decode('utf-8')
8     print(f"Received Request:\n{request_data}")
9
```

```
10     # Close the client socket
11     client_socket.close()
```

3. Sending a Response

```
1  while True:
2      # Accept incoming client connections
3      client_socket, client_address = server_socket.accept()
4      print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
5
6      # Receive and print the client's request data
7      request_data = client_socket.recv(1024).decode('utf-8')
8      print(f"Received Request:\n{request_data}")
9
10     # Prepare and send a simple HTTP response
11     response = "HTTP/1.1 200 OK\nContent-Type: text/html\n\nHello, World!"
12     client_socket.send(response.encode('utf-8'))
13
14     # Close the client socket
15     client_socket.close()
```

4. Request Handling (Basic Routing)

```
1  while True:
2      # Accept incoming client connections
3      client_socket, client_address = server_socket.accept()
4      print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
5
6      # Receive and print the client's request data
7      request_data = client_socket.recv(1024).decode('utf-8')
8      print(f"Received Request:\n{request_data}")
9
10     # Parse the request to get the HTTP method and path
11     request_lines = request_data.split('\n')
12     request_line = request_lines[0].strip().split()
13     method = request_line[0]
14     path = request_line[1]
15
16     # Prepare and send an appropriate HTTP response based on the path
17     if path == '/':
18         response_content = 'Hello, World!'
19     else:
20         response_content = 'Page not found.'
21
22     response = f'HTTP/1.1 200 OK\nContent-Type: text/html\n\n{response_content}'
23     client_socket.send(response.encode('utf-8'))
24
25     # Close the client socket
26     client_socket.close()
```

5. Signal Handling (Ctrl+C or Kill Signals)

```
1  import signal
2  import sys
3
```

```

4 # Function to handle Ctrl+C and other signals
5 def signal_handler(sig, frame):
6     print("\nShutting down the server...")
7     server_socket.close()
8     sys.exit(0)
9
10 # Register the signal handler
11 signal.signal(signal.SIGINT, signal_handler)
12
13 while True:
14     # Accept incoming client connections
15     client_socket, client_address = server_socket.accept()
16     print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
17
18     # Receive and print the client's request data
19     request_data = client_socket.recv(1024).decode('utf-8')
20     print(f"Received Request:\n{request_data}")
21
22     # Parse the request to get the HTTP method and path
23     request_lines = request_data.split('\n')
24     request_line = request_lines[0].strip().split()
25     method = request_line[0]
26     path = request_line[1]
27
28     # Prepare and send an appropriate HTTP response based on the path
29     if path == '/':
30         response_content = 'Hello, World!'
31     else:
32         response_content = 'Page not found.'
33
34     response = f'HTTP/1.1 200 OK\nContent-Type: text/html\n\n{response_content}'
35     client_socket.send(response.encode('utf-8'))
36
37     # Close the client socket
38     client_socket.close()

```

The web server gradually evolves from simply listening for connections to handling requests, sending responses, and adding signal handling for graceful server shutdown. It's important to note that this is a simplified example for educational purposes and lacks many features of a production-ready web server.

6. Add routing and 404 handling to the existing code

```

1 import socket
2 import signal
3 import sys
4 from time import sleep
5
6 # Define the server's IP address and port
7 HOST = '127.0.0.1' # IP address to bind to (localhost)
8 PORT = 8080       # Port to listen on
9
10 # Create a socket that uses IPv4 and TCP
11 server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 # Bind the socket to the address and port
14 server_socket.bind((HOST, PORT))
15

```

```

16 # Listen for incoming connections
17 server_socket.listen(5) # Backlog for multiple simultaneous connections
18 print(f"Server is listening on {HOST}:{PORT}")
19
20 # Function to handle Ctrl+C and other signals
21 def signal_handler(sig, frame):
22     print("\nShutting down the server...")
23     server_socket.close()
24     sys.exit(0)
25
26 # Register the signal handler
27 signal.signal(signal.SIGINT, signal_handler)
28
29 # Function to handle client requests
30 def handle_request(client_socket):
31     # Receive and print the client's request data
32     request_data = client_socket.recv(1024).decode('utf-8')
33     print(f"Received Request:\n{request_data}")
34
35     # Parse the request to get the HTTP method and path
36     request_lines = request_data.split('\n')
37     request_line = request_lines[0].strip().split()
38     method = request_line[0]
39     path = request_line[1]
40
41     # Initialize the response content and status code
42     response_content = ''
43     status_code = 200
44
45     # Define a simple routing mechanism
46     if path == '/':
47         sleep(15)
48         response_content = 'Hello, World!'
49     elif path == '/about':
50         response_content = 'This is the About page.'
51     else:
52         response_content = '404 Not Found'
53         status_code = 404
54
55     # Prepare the HTTP response
56     response = f'HTTP/1.1 {status_code} OK\nContent-Type: text/html\n\n{response_content}'
57     client_socket.send(response.encode('utf-8'))
58
59     # Close the client socket
60     client_socket.close()
61
62 while True:
63     # Accept incoming client connections
64     client_socket, client_address = server_socket.accept()
65     print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
66
67     try:
68         # Handle the client's request in a separate thread
69         handle_request(client_socket)
70     except KeyboardInterrupt:
71         # Handle Ctrl+C interruption here (if needed)
72         pass

```

1. We've added routing logic within the `handle_request` function. Depending on the path requested by the client, it responds with different content and sets an appropriate status code.
2. We've introduced a `status_code` variable to determine the HTTP response status code. If the requested path is not found, we set it to 404.
3. The `response_content` variable holds the content that will be sent in the HTTP response.
4. We've modified the response line to include the status code and provide a content type header.
5. In the `handle_request` function, if the requested path is not found, it responds with a "404 Not Found" message and sets the status code accordingly.

Now, your server includes basic routing and handles 404 errors gracefully.

7. Add multithreading for requests handling

```
1  import socket
2  import signal
3  import sys
4  import threading
5  from time import sleep
6
7  # Define the server's IP address and port
8  HOST = '127.0.0.1' # IP address to bind to (localhost)
9  PORT = 8080        # Port to listen on
10
11 # Create a socket that uses IPv4 and TCP
12 server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13
14 # Bind the socket to the address and port
15 server_socket.bind((HOST, PORT))
16
17 # Listen for incoming connections
18 server_socket.listen(5) # Increased backlog for multiple simultaneous connections
19 print(f"Server is listening on {HOST}:{PORT}")
20
21 # Function to handle client requests
22 def handle_request(client_socket):
23     # Receive and print the client's request data
24     request_data = client_socket.recv(1024).decode('utf-8')
25     print(f"Received Request:\n{request_data}")
26
27     # Parse the request to get the HTTP method and path
28     request_lines = request_data.split('\n')
29     request_line = request_lines[0].strip().split()
30     method = request_line[0]
31     path = request_line[1]
32
33     # Initialize the response content and status code
34     response_content = ''
35     status_code = 200
36
37     # Define a simple routing mechanism
38     if path == '/':
39         sleep(15)
40         response_content = 'Hello, World!'
41     elif path == '/about':
42         response_content = 'This is the About page.'
43     else:
```

```

44     response_content = '404 Not Found'
45     status_code = 404
46
47     # Prepare the HTTP response
48     response = f'HTTP/1.1 {status_code} OK\nContent-Type: text/html\n\n{response_content}'
49     client_socket.send(response.encode('utf-8'))
50
51     # Close the client socket
52     client_socket.close()
53
54 # Function to handle Ctrl+C and other signals
55 def signal_handler(sig, frame):
56     print("\nShutting down the server...")
57     server_socket.close()
58     sys.exit(0)
59
60 # Register the signal handler
61 signal.signal(signal.SIGINT, signal_handler)
62
63 while True:
64     # Accept incoming client connections
65     client_socket, client_address = server_socket.accept()
66     print(f"Accepted connection from {client_address[0]}:{client_address[1]}")
67
68     # Create a thread to handle the client's request
69     client_handler = threading.Thread(target=handle_request, args=(client_socket,))
70     client_handler.start()

```