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

```
# find process listening to port 8080
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:

```
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)

```python
import socket

# Define the server's IP address and port
HOST = '127.0.0.1'  # IP address to bind to (localhost)
PORT = 8080         # Port to listen on

# Create a socket that uses IPv4 and TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
server_socket.bind((HOST, PORT))

# Listen for incoming connections
server_socket.listen(5)  # Backlog for multiple simultaneous connections
print(f"Server is listening on {HOST}:{PORT}")

client_socket, client_address = server_socket.accept()
print(client_address)
```

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

```python
while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

```

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

3. Sending a Response

```
while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

    # Prepare and send a simple HTTP response
    response = "HTTP/1.1 200 OK\nContent-Type: text/html\n\nHello, World!"
    client_socket.send(response.encode('utf-8'))

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

4. Request Handling (Basic Routing)

```
while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

    # Parse the request to get the HTTP method and path
    request_lines = request_data.split('\n')
    request_line = request_lines[0].strip().split()
    method = request_line[0]
    path = request_line[1]

    # Prepare and send an appropriate HTTP response based on the path
    if path == '/':
        response_content = 'Hello, World!'
    else:
        response_content = 'Page not found.'

    response = f'HTTP/1.1 200 OK\nContent-Type: text/html\n\n{response_content}'
    client_socket.send(response.encode('utf-8'))

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

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

```
import signal
import sys

```

```python
# Function to handle Ctrl+C and other signals
def signal_handler(sig, frame):
    print("\nShutting down the server...")
    server_socket.close()
    sys.exit(0)

# Register the signal handler
signal.signal(signal.SIGINT, signal_handler)

while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

    # Parse the request to get the HTTP method and path
    request_lines = request_data.split('\n')
    request_line = request_lines[0].strip().split()
    method = request_line[0]
    path = request_line[1]

    # Prepare and send an appropriate HTTP response based on the path
    if path == '/':
        response_content = 'Hello, World!'
    else:
        response_content = 'Page not found.'

    response = f'HTTP/1.1 200 OK\nContent-Type: text/html\n\n{response_content}'
    client_socket.send(response.encode('utf-8'))

    # Close the client socket
    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

```python
import socket
import signal
import sys
from time import sleep

# Define the server's IP address and port
HOST = '127.0.0.1'  # IP address to bind to (localhost)
PORT = 8080         # Port to listen on

# Create a socket that uses IPv4 and TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
server_socket.bind((HOST, PORT))

```

```python
# Listen for incoming connections
server_socket.listen(5)  # Backlog for multiple simultaneous connections
print(f"Server is listening on {HOST}:{PORT}")

# Function to handle Ctrl+C and other signals
def signal_handler(sig, frame):
    print("\nShutting down the server...")
    server_socket.close()
    sys.exit(0)

# Register the signal handler
signal.signal(signal.SIGINT, signal_handler)

# Function to handle client requests
def handle_request(client_socket):
    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

    # Parse the request to get the HTTP method and path
    request_lines = request_data.split('\n')
    request_line = request_lines[0].strip().split()
    method = request_line[0]
    path = request_line[1]

    # Initialize the response content and status code
    response_content = ''
    status_code = 200

    # Define a simple routing mechanism
    if path == '/':
        sleep(15)
        response_content = 'Hello, World!'
    elif path == '/about':
        response_content = 'This is the About page.'
    else:
        response_content = '404 Not Found'
        status_code = 404

    # Prepare the HTTP response
    response = f'HTTP/1.1 {status_code} OK\nContent-Type: text/html\n\n{response_content}'
    client_socket.send(response.encode('utf-8'))

    # Close the client socket
    client_socket.close()

while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    try:
        # Handle the client's request in a separate thread
        handle_request(client_socket)
    except KeyboardInterrupt:
        # Handle Ctrl+C interruption here (if needed)
        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

```python
import socket
import signal
import sys
import threading
from time import sleep

# Define the server's IP address and port
HOST = '127.0.0.1'  # IP address to bind to (localhost)
PORT = 8080         # Port to listen on

# Create a socket that uses IPv4 and TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
server_socket.bind((HOST, PORT))

# Listen for incoming connections
server_socket.listen(5)  # Increased backlog for multiple simultaneous connections
print(f"Server is listening on {HOST}:{PORT}")

# Function to handle client requests
def handle_request(client_socket):
    # Receive and print the client's request data
    request_data = client_socket.recv(1024).decode('utf-8')
    print(f"Received Request:\n{request_data}")

    # Parse the request to get the HTTP method and path
    request_lines = request_data.split('\n')
    request_line = request_lines[0].strip().split()
    method = request_line[0]
    path = request_line[1]

    # Initialize the response content and status code
    response_content = ''
    status_code = 200

    # Define a simple routing mechanism
    if path == '/':
        sleep(15)
        response_content = 'Hello, World!'
    elif path == '/about':
        response_content = 'This is the About page.'
    else:
```

```
        response_content = '404 Not Found'
        status_code = 404

    # Prepare the HTTP response
    response = f'HTTP/1.1 {status_code} OK\nContent-Type: text/html\n\n{response_content}'
    client_socket.send(response.encode('utf-8'))

    # Close the client socket
    client_socket.close()

# Function to handle Ctrl+C and other signals
def signal_handler(sig, frame):
    print("\nShutting down the server...")
    server_socket.close()
    sys.exit(0)

# Register the signal handler
signal.signal(signal.SIGINT, signal_handler)

while True:
    # Accept incoming client connections
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # Create a thread to handle the client's request
    client_handler = threading.Thread(target=handle_request, args=(client_socket,))
    client_handler.start()
```