

# Course: Computer network

## Assignment 1 - Implement HTTP server and chat application

---

October 2, 2025

**Goal:** The objective of this assignment is the application of major components in a computer network, for example, client server paradigm, the peer-to-peer paradigm and the network programming.

**Content:** In detail, student will practice with three major modules: the client-server HTTP communication, the peer-to-peer based chat application and the TCP/IP connection, which includes

- client processes and server processes.
- multiple peer processes connect together.
- the socket TCP network programming.

Besides, student will practice the design and implementation of Simple peer-to-peer protocol via TCP/IP.

**Result:** After this assignment, student can understand partly the principle of a computer network system. They can understand and design the role of each types of processes, i.e server process, client process and tracker peer process in a network communication.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	An overview . . . . .	3
1.2	Source Code . . . . .	3
1.3	Server process . . . . .	4
1.4	How to Create a Server Process? . . . . .	4
1.5	How to Run the Server . . . . .	5
<b>2</b>	<b>Implementation</b>	<b>7</b>
2.1	HTTP server with cookie session . . . . .	7
2.2	Implement hybrid chat application . . . . .	8
2.3	Put It All Together . . . . .	9
<b>3</b>	<b>Submission</b>	<b>10</b>
3.1	Source code . . . . .	10
3.2	Requirements . . . . .	10
3.3	Grading . . . . .	10
3.4	Code of ethics . . . . .	10

# 1 Introduction

## 1.1 An overview

In this work, students implement a HTTP server application equipped with a webapp chat application. The server application has multiple server process deployments with a proxy server process and multiple backend server process. A WeApRous framework deployment will help provide the environment for implementing peer-to-peer application

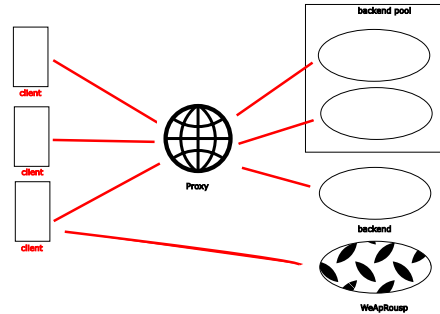


Figure 1: The general view of key modules in this assignment

Through those modules, the server framework allows multi-processes created by users to communicate and use the computing resources. .

## 1.2 Source Code

After downloading the source code of the assignment in the *Resource* section on the portal platform and extracting it, you will see the source code organized as follows.

- http\_daemon
  - daemon: response.py, request.py, backend.py, httpadapter.py, weaprous.py, dictionary.py
  - www: default location for static HTML webpages
    - \* index.html
  - static: default location for other HTML static objects
    - \* images: default location for images
    - \* css: default location for css file
    - \* js: default location for js
  - apps: default location for custom webapp
  - cert: default location for certification
  - db: default location for database
  - config: default location for proxy configuration file
  - start\_proxy.py: the main program to start a proxy process
  - start\_backend.py: the main program to start a backend process
  - start\_sampleapp.py: the main program to start a webapp (WeApRous) process

### 1.3 Server process

**Proxy** The proxy acts as an intermediary between the clients and the backend or webapp services. In this architecture, it receives incoming requests from multiple clients and forwards them to the appropriate backend resources. This process is crucial for managing traffic, optional for security policies, and enable load balancing/forwarding. By abstracting the backend infrastructure from the clients, the proxy helps ensure scalability and resilience, while also allowing for centralized control over routing and access.

**Backend** The backend refers to the core services or servers that handle the actual processing of client requests. In the diagram, the backend pool includes multiple backend instances, which may represent a set of processed within a logic layers. These components are responsible for executing and generating responses. The "previous step" proxy process routes the client requests to these backend units based on availability, load, or specific routing rules, ensuring efficient and reliable service delivery.

**WebApp - WeApRous** The WebApp, labeled as "WeApRous" (shorten form of WebAppRoutes) in the image, represents a specialized group of web-facing applications or services. These are typically user-representative components such as web servers or frontend applications that deliver content and interfaces to the clients. The WeApRous may be optimized for rendering HTML, serving static assets, or interacting with users directly through RESTful. Like the backend, it receives traffic through the proxy, which helps manage access and performance. WeApRous can be run in independent mode without proxy

### 1.4 How to Create a Server Process?

The content of each process is actually a copy of a program stored on disk. Thus to create a process, we must first generate the program which describes its content. A program is defined by a single file with the following format:

Listing 1: "how to create a proxy"

```
# From daemon/proxy.py
def create_proxy(ip, port, routes):
    run_proxy(ip, port, routes)
```

Listing 2: "Create a backend"

```
# From daemon/backend.py
def create_backend(ip, port, routes={}):
    run_backend(ip, port, routes)
```

Listing 3: "Create a webapp"

```
# From ./start_sampleapp.py
app = WeApRous()

@app.route('/op1', methods=['POST'])
5 def op1(headers="guest", body="anonymous"):
    ...

@app.route('/op2', methods=['PUT'])
10 def op2(headers="guest", body="anonymous"):
    ...
```

## 1.5 How to Run the Server

What we are going to do in this assignment is to implement a simple HTTP server. To start these server process, we must create a script that revokes these server process and provide the hardware and the environment that we will work. The description file is defined in the following format (adapt these IP socket address to your system environment):

```
# From config/proxy.conf
host "192.168.56.103:8080" {
    proxy_pass http://192.168.56.103:9000;
}
5 host "app1.local" {
    proxy_pass http://192.168.56.103:9001;
}
10 host "app2.local" {
    proxy_set_header Host $host;

    proxy_pass http://192.168.56.210:9002;
    proxy_pass http://192.168.56.220:9002;
15
    dist_policy round-robin
}
```

The **configuration basic structure** Each configuration block begins with a host declaration followed by a set of instructions enclosed in

- **host:** declares the target hostname or IP address and optional port.
- **Block {}:** contains directives that define how requests to this host should be handled.

### Supported directives

- **proxy\_pass [URL];** defines the backend server to which requests should be forwarded. You can specify multiple proxy\_pass entries to enable load balancing policy.
- **proxy\_set\_header [HeaderName] [Value];** sets custom headers for forwarded requests. This example alternate the original Host header field in the client requests.
- **dist\_policy [strategy];** Specifies the distribution policy when multiple backends are defined.
  - round-robin: distributes requests evenly across all backends.
  - Other strategies (if supported) might include least-conn, random, etc.

Listing 4: "Start the processes"

```
# Entry point for launching the server. This block parses command-line
# arguments to determine the server's IP address and port.
# :arg --server-ip (str): IP address to bind the server (default: 127.0.0.1).
# :arg --server-port (int): Port number to bind the server (default: 9000).
5 parser = argparse.ArgumentParser(prog='Proxy', description='', epilog='Proxy daemon')
parser.add_argument('--server-ip', default='0.0.0.0')
```

```
    parser.add_argument('--server-port', type=int, default=PROXY_PORT)

    args = parser.parse_args()
10    ip = args.server_ip
    port = args.server_port

    # From start_proxy.py
    from daemon import create_proxy
15    routes = parse_virtual_hosts("config/proxy.conf")
    create_proxy(ip, port, routes)

    # From start_backend.py
20    from daemon import create_backend

    create_backend(ip, port)

    # From start_sampleapp.py
25    from daemon.weaprous import WeApRous

    # Prepare and launch the RESTful application
    app.prepare_address(ip, port)
    app.run()
```

## 2 Implementation

### 2.1 HTTP server with cookie session

**Description** Implement cookie-based session handling in a basic HTTP server using Python's socket and threading modules. The goal is to authenticate users via a login form and maintain their login state using HTTP cookies, allowing access to protected resources (e.g., the index page) only if authenticated.

**Notes** it is recommended to use incognito tab since HTTP apply conditional GET by default. The browser cache can be cleared to perform the system experiment properly.

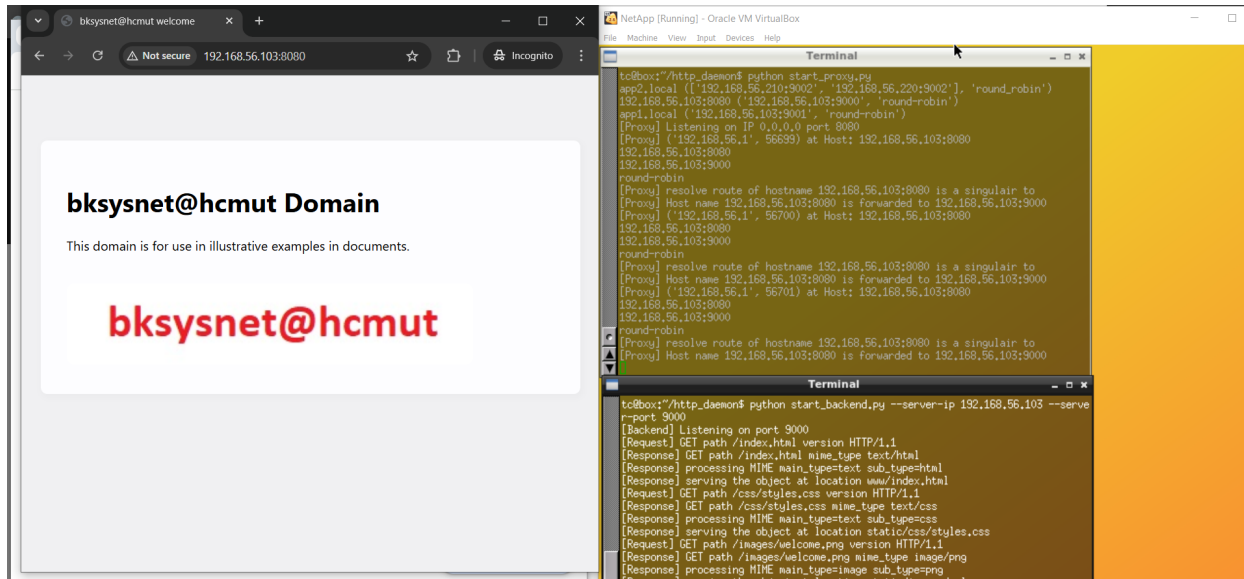


Figure 2: The screenshot of starting a proxy and backends

In Figure 3, a proxy is started at port 8080 and a backend is started at port 9000. User browser surf the webpage at `http://IP:8080` and is redirected to be served at backend `http://IP:9000` address

#### Task 1A: Implement authentication handling

- On receiving a POST request to `/login`, the server must validate the submitted credentials (username=admin, password=password).
- If valid, the server must respond with the index page and include a Set-Cookie: auth=true header to indicate successful login.
- If invalid, the server must respond with a 401 Unauthorized page.

#### Task 1B: Implement cookie-based access control

- When a client sends a GET request to `/`, the server must check for the presence of the auth=true cookie.
- If the cookie is present, the server serves the index page.
- If the cookie is missing or incorrect, the server responds with 401 Unauthorized page

**Task requirements**

- Header Parsing
- Session Management
- Concurrency
- Error Handling

**2.2 Implement hybrid chat application**

**Description** Develop a hybrid network application that has a chat system (similar to skype), combining both client-server and peer-to-peer (P2P) paradigms. The application must support channel management, and synchronization across distributed peers.

**Equipped webapp - WeApRous** The webapp support RESTful (Representational State Transfer) is an architectural style for designing networked applications. It relies on standard HTTP methods [GET], [POST], [PUT], [DELETE] to perform operations on resources, which are typically represented as URLs. RESTful APIs are stateless, scalable, and easy to integrate across platforms. Custom route URLs allow developers to define meaningful and readable endpoints. These routes improve clarity and make your API friendly to handle more complex logic including query parameters or nested paths.

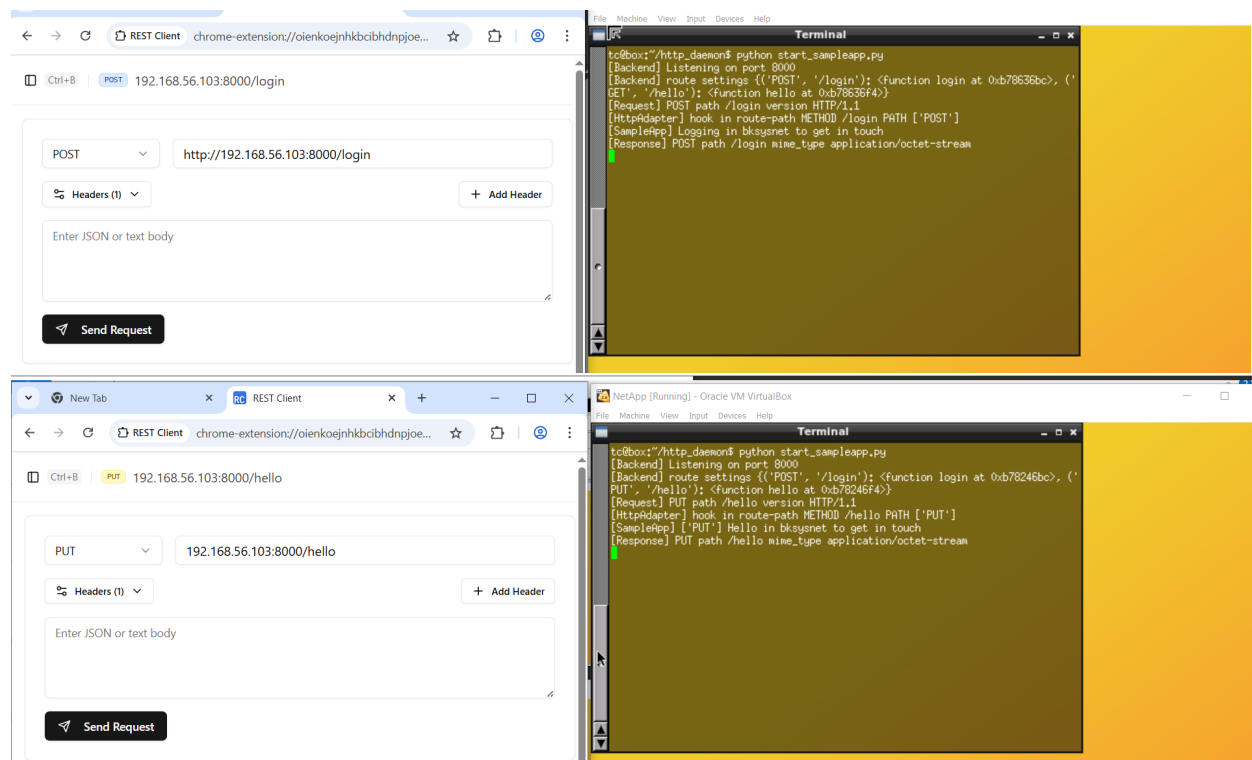


Figure 3: The screenshot of webapp (WeApRous) serves request at [PUT]/login and [POST]/hello

In Figure 3, a user webapp is started at port `http://IP:8000` with two handler at method [PUT] path `"/login"` and method [POST] path `"/hello"`. The handler is installed by user webapp and is pres

**Core functional requirements** the application has these following requirements

**Initialization phase (Client-Server paradigm)**

- *Peer registration*: When a new peer joins, it must submit its IP and port to the centralized server.
- *Tracker update*: The centralized server must maintain a tracking list of active peers.
- *Peer discovery*: Peers can request the current list of active peers from the server.
- *Connection setup*: Peers use the tracking list to initiate direct P2P connections.

#### Peer chatting phase (Peer-to-Peer Paradigm)

- *Broadcast connection*: A peer must broadcast messages to all connected peers.
- *Direct peer communication*: Peers exchange messages without routing through the centralized server during live sessions.

#### Channel management

- *Channel listing*: Users can view channels they've joined.
- *Message display*: Each channel has a scrollable message window.
- *Message submission*: UI must support text input and submission.
- *No edit/delete*: Messages are immutable once sent.
- *Notification system*: Users must be notified when new messages arrive.
- *Access control (Optional)*: Channels may define custom access policies.

#### Task requirements

- Client-server process programming
- Protocol design in message communication and the processing procedure
  - **login API** example `http://IP:port/login/`
  - **submit-info API** example `http://IP:port/submit-info/`
  - **add-list API** example `http://IP:port/add-list/`
  - **get-list API** example `http://IP:port/get-list/`
  - **connect-peer API** example `http://IP:port/connect-peer/`
  - **broadcast-peer API** example `http://IP:port/broadcast-peer/`
  - **send-peer API** example `http://IP:port/send-peer/`
- Concurrency
- Error Handling

## 2.3 Put It All Together

Finally, we combine both HTTP Cookies work and WeApRous derivations to form a complete server application.

## 3 Submission

### 3.1 Source code

**Requirement:** you have to code the HTTP server and application followed by the coding style. References:

- PEP 8 – Style Guide for Python Code - <https://peps.python.org/pep-0008> and
- PEP 257 – Docstring Conventions - <https://peps.python.org/pep-0257/>

### 3.2 Requirements

**Application and protocol design** implement the design based on TCP/IP protocol is mandatory.

*\*\*Note\*\* The course outcomes indicate the ability of independently develop a software architecture and network protocol. Students are provided a sample network application and are not permitted to use other existing web framework. The framework must be built by yourself.*

**Cookies session** implement the authentication handling and the cookies access control subsystem2.1.

**Chat application** implement the hybrid chat application.

After you finish the assignment, move your report to source code directory and compress the entire directory into a single file named `assignment_STUDENTID.zip` and submit to LMS.

### 3.3 Grading

You must complete this assignment in groups of 4 or 5 students. The overall grade for your group is determined by the following components:

- **Demonstration (7 points)**
  - Cookies: 3 points
  - ChatApp Client-server paradigm: 2 points
  - ChatApp Peer-to-peer paradigm: 2 points
- **Report (3 points)**

### 3.4 Code of ethics

Faculty staff members involved in code development reserved all the copyright of the project source code.

**Source Code License Grant:** Author(s) hereby grant(s) to Licensee personal permission to use and modify the Licensed Source Code for the sole purpose of studying while attending the course CO3093/CO3094 at HCMUT.

## Revision History

Revision	Date	Author(s)	Description
1.0	01.2025	pdnguyen,	Initialize HTTP server, WeApRous Webapp framework

start\_proxy  
~~~~~

This module serves as the entry point for launching a proxy server using Python's socket framework. It parses command-line arguments to configure the server's IP address and port, reads virtual host definitions from a configuration file, and initializes the proxy server with routing information.

Requirements:  
-----

- socket: provide socket networking interface.
- threading: enables concurrent client handling via threads.
- argparse: parses command-line arguments for server configuration.
- re: used for regular expression matching in configuration parsing
- response: response utilities.
- httpadapter: the class for handling HTTP requests.
- urlparse: parses URLs to extract host and port information.
- daemon.create\_proxy: initializes and starts the proxy server.

## Modules

[argparse](#)

[re](#)

[socket](#)

[threading](#)

## Functions

**parse\_virtual\_hosts(filepath)**  
Parses virtual host blocks from a config file.

:filepath (str): Path to the NGINX config file.  
:rtype list of dict: Each dict contains 'listen'and 'server\_name'.

## Data

**PROXY\_PORT** = 8080

start\_backend  
~~~~~

This module provides a simple entry point for deploying backend server process using the socket framework. It parses command-line arguments to configure the server's IP address and port, and then launches the backend server.

Modules

[argparse](#)[socket](#)

Data

**PORT** = 9000

start\_sampleapp  
~~~~~

This module provides a sample RESTful web application using the WeApRous framework.

It defines basic route handlers and launches a TCP-based backend server to serve HTTP requests. The application includes a login endpoint and a greeting endpoint, and can be configured via command-line arguments.

## Modules

[argparse](#)[json](#)[socket](#)

## Functions

**hello**(headers, body)  
Handle greeting via GET request.

This route prints a greeting message to the console using the provided headers and body.

:param headers (str): The request headers or user identifier.  
:param body (str): The request body or message payload.

**login**(headers='guest', body='anonymous')  
Handle user login via POST request.

This route simulates a login process and prints the provided headers and body to the console.

:param headers (str): The request headers or user identifier.  
:param body (str): The request body or login payload.

## Data

**PORT** = 8000  
**app** = <daemon.weaprous.WeApRous instance>

# proxy

daemon.proxy

This module implements a simple proxy server using Python's socket and threading libraries. It routes incoming HTTP requests to backend services based on hostname mappings and returns the corresponding responses to clients.

Requirement:

- socket: provides socket networking interface.
- threading: enables concurrent client handling via threads.
- response: customized :class: `Response <Response>` utilities.
- httpadapter: :class: `HttpAdapter <HttpAdapter >` adapter for HTTP request processing.
- dictionary: :class: `CaseInsensitiveDict <CaseInsensitiveDict>` for managing headers and cookies.

## Modules

[socket](#)

[threading](#)

## Functions

### **create\_proxy(ip, port, routes)**

Entry point for launching the proxy server.

:params ip (str): IP address to bind the proxy server.  
 :params port (int): port number to listen on.  
 :params routes (dict): dictionary mapping hostnames and location.

### **forward\_request(host, port, request)**

Forwards an HTTP request to a backend server and retrieves the response.

:params host (str): IP address of the backend server.  
 :params port (int): port number of the backend server.  
 :params request (str): incoming HTTP request.

:rtype bytes: Raw HTTP response from the backend server. If the connection fails, returns a 404 Not Found response.

### **handle\_client(ip, port, conn, addr, routes)**

Handles an individual client connection by parsing the request, determining the target backend, and forwarding the request.

The handler extracts the Host header from the request to matches the hostname against known routes. In the matching condition, it forwards the request to the appropriate backend.

The handler sends the backend response back to the client or returns 404 if the hostname is unreachable or is not recognized.


:params ip (str): IP address of the proxy server.  
 :params port (int): port number of the proxy server.  
 :params conn (socket.socket): client connection socket.  
 :params addr (tuple): client address (IP, port).  
 :params routes (dict): dictionary mapping hostnames and location.

### **run\_proxy(ip, port, routes)**

Starts the proxy server and listens for incoming connections.

The process binds the proxy server to the specified IP and port. In each incoming connection, it accepts the connections and spawns a new thread for each client using `handle\_client`.

:params ip (str): IP address to bind the proxy server.



```
:params port (int): port number to listen on.  
:params routes (dict): dictionary mapping hostnames and location.
```

# backend

daemon.backend

~~~~~

This module provides a backend object to manage and persist backend daemon. It implements a basic backend server using Python's socket and threading libraries. It supports handling multiple client connections concurrently and routing requests using a custom HTTP adapter.

Requirements:

-----

- socket: provide socket networking interface.
- threading: Enables concurrent client handling via threads.
- response: response utilities.
- httpadapter: the class for handling HTTP requests.
- CaseInsensitiveDict: provides dictionary for managing headers or routes.

Notes:

-----

- The server create daemon threads for client handling.
- The current implementation error handling is minimal, socket errors are printed to the console.
- The actual request processing is delegated to the HttpAdapter class.

Usage Example:

-----

```
>>> create\_backend("127.0.0.1", 9000, routes={})
```

## Modules

[argparse](#)

[socket](#)

[threading](#)

## Functions

**create\_backend**(ip, port, routes={})

Entry point for creating and running the backend server.

:param ip (str): IP address to bind the server.

:param port (int): Port number to listen on.

:param routes (dict, optional): Dictionary of route handlers. Defaults to empty dict.

**handle\_client**(ip, port, conn, addr, routes)

Initializes an HttpAdapter instance and delegates the client handling logic to it.

:param ip (str): IP address of the server.

:param port (int): Port number the server is listening on.

:param conn (socket.socket): Client connection socket.

:param addr (tuple): client address (IP, port).

:param routes (dict): Dictionary of route handlers.

**run\_backend**(ip, port, routes)

Starts the backend server, binds to the specified IP and port, and listens for incoming connections. Each connection is handled in a separate thread. The backend accepts incoming connections and spawns a thread for each client.

:param ip (str): IP address to bind the server.

:param port (int): Port number to listen on.

:param routes (dict): Dictionary of route handlers.

# request

daemon.request

~~~~~

This module provides a [Request](#) object to manage and persist request settings (cookies, auth, proxies).

## Classes

### [Request](#)

#### class **Request**

The fully mutable :class:`[Request](#) <[Request](#)>` object, containing the exact bytes that will be sent to the server.

Instances are generated from a :class:`[Request](#) <[Request](#)>` object, and should not be instantiated manually; doing so may produce undesirable effects.

Usage::

```
>>> import daemon.request
>>> req = request.Request()
## Incoming message obtain aka. incoming_msg
>>> r = req.prepare(incoming_msg)
>>> r
<Request>
```

Methods defined here:

**`__init__(self)`**

**`extract_request_line(self, request)`**

**`prepare(self, request, routes=None)`**  
Prepares the entire request with the given parameters.

**`prepare_auth(self, auth, url="")`**

**`prepare_body(self, data, files, json=None)`**

**`prepare_content_length(self, body)`**

**`prepare_cookies(self, cookies)`**

**`prepare_headers(self, request)`**  
Prepares the given HTTP headers.

---

Data and other attributes defined here:

**`__attrs__`** = ['method', 'url', 'headers', 'body', 'reason', 'cookies', 'body', 'routes', 'hook']

# response

daemon.response

This module provides a :class: `Response` object to manage and persist response settings (cookies, auth, proxies), and to construct HTTP responses based on incoming requests.

The current version supports MIME type detection, content loading and header formatting

## Modules

[datetime](#)

[mimetypes](#)

[os](#)

## Classes

### [Response](#)

#### class **Response**

The :class: `Response` object, which contains a server's response to an HTTP request.

Instances are generated from a :class: `Request` object, and should not be instantiated manually; doing so may produce undesirable effects.

:class: `Response` object encapsulates headers, content, status code, cookies, and metadata related to the request-response cycle. It is used to construct and serve HTTP responses in a custom web server.

:attr: `status_code` (int): HTTP status code (e.g., 200, 404).  
:attr: `headers` (dict): dictionary of response headers.  
:attr: `url` (str): url of the response.  
:attr: `encoding` (str): encoding used for decoding response content.  
:attr: `history` (list): list of previous `Response` objects (for redirects).  
:attr: `reason` (str): textual reason for the status code (e.g., "OK", "Not Found").  
:attr: `cookies` (CaseInsensitiveDict): response cookies.  
:attr: `elapsed` (datetime.timedelta): time taken to complete the request.  
:attr: `request` (PreparedRequest): the original request object.

Usage::

```
>>> import Response
>>> resp = Response()
>>> resp.build_response(req)
>>> resp
<Response>
```

Methods defined here:

**\_\_init\_\_**(self, request=None)

Initializes a new :class: `Response` object.

: params request : The originating request object.

**build\_content**(self, path, base\_dir)

Loads the objects file from storage space.

: params path (str): relative path to the file.

: params base\_dir (str): base directory where the file is located.

:rtype tuple: (int, bytes) representing content length and content data.

**build\_notfound(self)**

Constructs a standard 404 Not Found HTTP response.

:rtype bytes: Encoded 404 response.

**build\_response(self, request)**

Builds a full HTTP response including headers and content based on the request.

:params request (class:`Request <Request>`): incoming request object.

:rtype bytes: complete HTTP response using prepared headers and content.

**build\_response\_header(self, request)**

Constructs the HTTP response headers based on the class:`Request <Request>` and internal attributes.

:params request (class:`Request <Request>`): incoming request object.

:rtypes bytes: encoded HTTP response header.

**get\_mime\_type(self, path)**

Determines the MIME type of a file based on its path.

"params path (str): Path to the file.

:rtype str: MIME type string (e.g., 'text/html', 'image/png').

**prepare\_content\_type(self, mime\_type='text/html')**

Prepares the Content-Type header and determines the base directory for serving the file based on its MIME type.

:params mime\_type (str): MIME type of the requested resource.

:rtype str: Base directory path for locating the resource.

:raises ValueError: If the MIME type is unsupported.

---

Data and other attributes defined here:

```
__attrs__ = ['_content', '_header', 'status_code', 'method', 'headers', 'url', 'history', 'encoding', 'reason', 'cookies', 'elapsed', 'request', 'body', 'reason']
```

## Data

```
BASE_DIR = "
```

```
# Copyright (C) 2025 pdnguyen of HCMC University of Technology VNU-HCM.  
# All rights reserved.  
# This file is part of the C03093/C03094 course.  
#  
# WeApRous release  
#  
# The authors hereby grant to Licensee personal permission to use  
# and modify the Licensed Source Code for the sole purpose of studying  
# while attending the course  
#
```

Classes

[\\_abcoll.MutableMapping](#)([\\_abcoll.Mapping](#))  
[CaseInsensitiveDict](#)

class **CaseInsensitiveDict**([\\_abcoll.MutableMapping](#))  
The :class:`[CaseInsensitiveDict](#)<[MutableMapping](#)>` object, which contains a custom behavior of MutuableMapping.

```
Usage::  
  
>>> import tools  
>>> word = CaseInsensitiveDict(status_code='404', msg="Not found")  
>>> code = word['status_code']  
>>> code  
404  
  
>>> msg = word['msg']  
>>> s.send(r)  
Not found  
  
>>> print(word)  
{'status_code': '404', 'msg': 'Not found'}
```

Method resolution order:  
[CaseInsensitiveDict](#)  
[\\_abcoll.MutableMapping](#)  
[\\_abcoll.Mapping](#)  
[\\_abcoll.Sized](#)  
[\\_abcoll.Iterable](#)  
[\\_abcoll.Container](#)  
[\\_builtin\\_.object](#)

Methods defined here:

```
__delitem__(self, key)  
__getitem__(self, key)  
__init__(self, *args, **kwargs)  
__iter__(self)  
__len__(self)  
__setitem__(self, key, value)
```

Data and other attributes defined here:

**`__abstractmethods__`** = frozenset([])

---

Methods inherited from [\\_abcoll.MutableMapping](#):

**`clear`**(self)

D.[clear](#)() -> None. Remove all items from D.

**`pop`**(self, key, default=<object object>)

D.[pop](#)(k[,d]) -> v, remove specified key and return the corresponding value.  
If key is not found, d is returned if given, otherwise KeyError is raised.

**`popitem`**(self)

D.[popitem](#)() -> (k, v), remove and return some (key, value) pair  
as a 2-tuple; but raise KeyError if D is empty.

**`setdefault`**(self, key, default=None)

D.[setdefault](#)(k[,d]) -> D.[get](#)(k,d), also set D[k]=d if k not in D

**`update`**(\*args, \*\*kwds)

D.[update](#)([E, ]\*\*F) -> None. Update D from mapping/iterable E and F.  
If E present and has a [.keys\(\)](#) method, does: for k in E: D[k] = E[k]  
If E present and lacks [.keys\(\)](#) method, does: for (k, v) in E: D[k] = v  
In either case, this is followed by: for k, v in F.[items](#)(): D[k] = v

---

Methods inherited from [\\_abcoll.Mapping](#):

**`__contains__`**(self, key)

**`__eq__`**(self, other)

**`__ne__`**(self, other)

**`get`**(self, key, default=None)

D.[get](#)(k[,d]) -> D[k] if k in D, else d. d defaults to None.

**`items`**(self)

D.[items](#)() -> list of D's (key, value) pairs, as 2-tuples

**`iteritems`**(self)

D.[iteritems](#)() -> an iterator over the (key, value) items of D

**`iterkeys`**(self)

D.[iterkeys](#)() -> an iterator over the keys of D

**`itervalues`**(self)

D.[itervalues](#)() -> an iterator over the values of D

**`keys`**(self)

D.[keys](#)() -> list of D's keys

**`values`**(self)

D.[values](#)() -> list of D's values

---

Data and other attributes inherited from [\\_abcoll.Mapping](#):

**`__hash__`** = None

---

Class methods inherited from [\\_abcoll.Sized](#):

**`__subclasshook__`**(cls, C) from [abc.ABCMeta](#)

---

Data descriptors inherited from [\\_abcoll.Sized](#):

**`__dict__`**

dictionary for instance variables (if defined)

**`__weakref__`**

list of weak references to the object (if defined)

---

Data and other attributes inherited from [`abcoll.Sized`](#):

**`__metaclass__`** = <class 'abc.ABCMeta'>

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as 'virtual subclasses' -- these and their descendants will be considered subclasses of the registering ABC by the built-in `issubclass()` function, but the registering ABC won't show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via `super()`).

# httpadapter

daemon.httpadapter

This module provides a http adapter object to manage and persist http settings (headers, bodies). The adapter supports both raw URL paths and RESTful route definitions, and integrates with Request and Response objects to handle client-server communication.

## Classes

### [HttpAdapter](#)

#### class **HttpAdapter**

A mutable :class:`HTTP adapter <HTTP adapter>` for managing client connections and routing requests.

The [`HttpAdapter`](#) class encapsulates the logic for receiving HTTP requests, dispatching them to appropriate route handlers, and constructing responses. It supports RESTful routing via hooks and integrates with :class:`Request <Request>` and :class:`Response <Response>` objects for full request lifecycle management.

Attributes:

ip (str): IP address of the client.  
port (int): Port number of the client.  
conn (socket): Active socket connection.  
connaddr (tuple): Address of the connected client.  
routes (dict): Mapping of route paths to handler functions.  
request (Request): Request object for parsing incoming data.  
response (Response): Response object for building and sending replies.

Methods defined here:

**`__init__(self, ip, port, conn, connaddr, routes)`**

Initialize a new [HttpAdapter](#) instance.

:param ip (str): IP address of the client.  
:param port (int): Port number of the client.  
:param conn (socket): Active socket connection.  
:param connaddr (tuple): Address of the connected client.  
:param routes (dict): Mapping of route paths to handler functions.

**`add_headers(self, request)`**

Add headers to the request.

This method is intended to be overridden by subclasses to inject custom headers. It does nothing by default.

:param request: :class:`Request <Request>` to add headers to.

**`build_proxy_headers(self, proxy)`**

Returns a dictionary of the headers to add to any request sent through a proxy.

:class:`[HttpAdapter](#) <HttpAdapter>`.

:param proxy: The url of the proxy being used for this request.  
:rtype: dict

**`build_response(self, req, resp)`**

Builds a :class:`Response <Response>` object

:param req: The :class:`Request <Request>` used to generate the response.

:param resp: The response object.  
:rtype: Response

### **handle\_client**(self, conn, addr, routes)

Handle an incoming client connection.

This method reads the request from the socket, prepares the request object, invokes the appropriate route handler if available, builds the response, and sends it back to the client.

:param conn (socket): The client socket connection.  
:param addr (tuple): The client's address.  
:param routes (dict): The route mapping for dispatching requests.

---

Data descriptors defined here:

### **extract\_cookies**

Build cookies from the :class:`Request <Request>` headers.

:param req:(Request) The :class:`Request <Request>` object.  
:param resp: (Response) The res:class:`Response <Response>` object.  
:rtype: cookies - A dictionary of cookie key-value pairs.

---

Data and other attributes defined here:

**\_\_attrs\_\_** = ['ip', 'port', 'conn', 'connaddr', 'routes', 'request', 'response']

daemon.weaprous  
~~~~~

This module provides a [WeApRous](#) object to deploy RESTful url web app with routing

Classes

[WeApRous](#)

class **WeApRous**

The fully mutable :class:`[WeApRous](#) <[WeApRous](#)>` object, which is a lightweight, mutable web application router for deploying RESTful URL endpoints.

The `[WeApRous](#)` class provides a decorator-based routing system for building simple RESTful web applications. The class allows developers to register route handlers using decorators and launch a TCP-based backend server to serve RESTful requests. Each route is mapped to a handler function based on HTTP method and path. It mappings supports tracking the combined HTTP methods and path route mappings internally.

```
Usage::
>>> import daemon.weaprous
>>> app = WeApRous()
>>> @app.route('/login', methods=['POST'])
>>> def login(headers="guest", body="anonymous"):
>>>     return {'message': 'Logged in'}

>>> @app.route('/hello', methods=['GET'])
>>> def hello(headers, body):
>>>     return {'message': 'Hello, world!'}

>>> app.run()
```

Methods defined here:

- [\\_\\_init\\_\\_](#)**(self)  
Initialize a new [WeApRous](#) instance.  
  
Sets up an empty route registry and prepares placeholders for IP and port.
- [prepare\\_address](#)**(self, ip, port)  
Configure the IP address and port for the backend server.  
  
:param ip (str): The IP address to bind the server.  
:param port (str): The port number to listen on.
- [route](#)**(self, path, methods=['GET'])  
Decorator to register a route handler for a specific path and HTTP methods.  
  
:param path (str): The URL path to route.  
:param methods (list): A list of HTTP methods (e.g., ['GET', 'POST']) to bind.  
  
:rtype: function - A decorator that registers the handler function.
- [run](#)**(self)  
Start the backend server and begin handling requests.  
  
This method launches the TCP server using the configured IP and port, and dispatches incoming requests to the registered route handlers.  
  
:raise: Error if IP or port has not been configured.