Server Library Documentation

July 14, 2024

1 Introduction

The `WebServer` module provides a robust framework for creating and managing a web server with features such as streaming responses, asynchronous request processing, and request routing. It is designed to handle HTTP GET and POST requests efficiently, with support for request logging and authorization.

2 Usage Example

Below is an example of how to use the `WebServer` module to set up and run a web server.

```
from webserver import WebServer
from test_text import text
from app.context_manager.context_manager import ServerContextManager

def main():
    port = 8080
    webserver = WebServer(port=port)

    @webserver.router.add('/needs_auth', 'GET', needs_auth=True)
    def needs_auth():
        return "This is a response for a request that needs authentication"

    @webserver.router.add('/sample_post', 'POST', needs_auth=False)
    def handle_post():
        return "This is a POST response"
    webserver.listen()

if __name__ == "__main__":
    main()
```

2.1 Explanation of the Example Code

- Routes are added using the `@webserver.router.add` decorator. Each route specifies the URL, HTTP method, and whether authentication is required.
- The `needs_auth` parameter in the route decorator determines if a route requires authentication.
- The server is run by creating an instance of the `WebServer` class and calling its `listen()` method.

3 Module Contents

3.1 WebServer Submodule

The `WebServer` submodule provides the main server configuration and management functionalities.

3.1.1 `singleton` Decorator

The `singleton` decorator ensures that only one instance of a class is created. If an instance of the class already exists, it returns that instance instead of creating a new one.

3.1.2 `WebServer` Class

The `WebServer` class encapsulates the server configuration and provides methods to start and stop the server.

Initialization

- `Inputs:`
  - `host` (str): The hostname or IP address where the server will be hosted. Default is `'127.0.0.1'`.
  - `port` (int): The port number on which the server will listen. Default is `8080`.
- `Outputs:` None
- `Responsibilities:` Initializes the server with the specified host and port, and sets up the router and listener components.

Attributes

- `router`: An instance of the `Router` class that handles request routing.
- `port`: The port number on which the server listens.
- `host`: The hostname or IP address where the server is hosted.
- `listener`: An instance of the `Listener` class that handles incoming connections.

Methods

- `listen()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Starts the server and begins listening for incoming connections.
- `close()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Stops the server and closes any open connections.

3.2 Router Submodule

The `Router` submodule provides a flexible and powerful mechanism for routing HTTP requests within a web server.

3.2.1 `Route` Class

The `Route` class represents an individual route in the router.

Initialization

- `Inputs:`
  - `URL` (str): The URL pattern for the route.
  - `method` (str): The HTTP method for the route (e.g., GET, POST).
  - `needs_auth` (bool): Indicates whether the route requires authentication.
- `Outputs:` None
- `Responsibilities:` Initializes a route with the specified URL, method, and authentication requirement.

3.2.2 `Router` Class

The `Router` class manages the collection of routes and processes incoming requests.

Initialization

- `Inputs:`
  - `request_queue` (list): A list to hold incoming requests for processing. Default is an empty list.
- `Outputs:` None
- `Responsibilities:` Initializes the router with an optional request queue.

Attributes

- `routes`: A dictionary holding the registered routes.
- `request_queue`: A list of incoming requests to be processed.

Methods

- `add(URL, method, needs_auth)`:
  - `Inputs:`
    - `URL` (str): The URL pattern for the route.
    - `method` (str): The HTTP method for the route.
    - `needs_auth` (bool): Indicates whether the route requires authentication.
  - `Outputs:` None
  - `Responsibilities:` Registers a new route with the router.
- `add_request(request)`:
  - `Inputs:`
    - `request` (str): The raw HTTP request text.
  - `Outputs:` None
  - `Responsibilities:` Adds a new request to the request queue.
- `parse_request_text(request)`:
  - `Inputs:`
    - `request` (str): The raw HTTP request text.
  - `Outputs:`

        * `http_request_line` (dict): Parsed HTTP request line containing method, path, and HTTP version.
        * `headers` (dict): Parsed HTTP headers.
        * `body` (dict or str): Parsed HTTP body.
    - `Responsibilities:` Parses the raw HTTP request text into its components.

- `process_request()`:
    - `Inputs:` None
    - `Outputs:`
        * `http_request_line` (dict): Parsed HTTP request line containing method, path, and HTTP version.
        * `headers` (dict): Parsed HTTP headers.
        * `body` (dict or str): Parsed HTTP body.
        * `request` (str): The original raw HTTP request text.
    - `Responsibilities:` Processes the next request in the request queue.

3.3 Listener Submodule

The `Listener` submodule is responsible for handling incoming connections to the web server.

3.3.1 `Listener` Class

The `Listener` class manages incoming connections and delegates them to the appropriate router for processing.

Initialization

- `Inputs:`
    - `host` (str): The hostname or IP address where the server will be hosted.
    - `port` (int): The port number on which the server will listen.
    - `router` (Router): An instance of the `Router` class to handle request routing.
- `Outputs:` None
- `Responsibilities:` Initializes the listener with the specified host, port, and router.

Attributes

- `host`: The hostname or IP address where the server is hosted.
- `port`: The port number on which the server listens.
- `router`: An instance of the `Router` class that handles request routing.

Methods

- `listen()`:
    - `Inputs:` None
    - `Outputs:` None
    - `Responsibilities:` Starts listening for incoming connections.
- `close()`:
    - `Inputs:` None
    - `Outputs:` None
    - `Responsibilities:` Stops listening and closes all connections.

3.4 `Listener` Class

The `Listener` class manages incoming connections and delegates them to the appropriate router for processing.

Initialization

- `Inputs:`
  - `host` (str): The hostname or IP address where the server will be hosted. Default is `'127.0.0.1'`.
  - `port` (int): The port number on which the server will listen. Default is `8080`.
  - `router` (Router): An instance of the `Router` class to handle request routing. Default is a new `Router` instance.
- `Outputs:` None
- `Responsibilities:` Initializes the listener with the specified host, port, and router.

Attributes

- `host`: The hostname or IP address where the server is hosted.
- `port`: The port number on which the server listens.
- `router`: An instance of the `Router` class that handles request routing.

Methods

- `listen()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Starts the server, binds to the specified host and port, and begins listening for incoming connections.
- `connect_to_client()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Asynchronously handles the connection to a client, receives requests, and sends appropriate responses.
- `recieve_requests(router)`:
  - `Inputs:`
    - `router` (Router): The router instance to handle the received request.
  - `Outputs:`
    - `request` (str): The received HTTP request.
    - `val` (bool): A flag indicating the validity of the request.
    - `message` (str): A message indicating the result of the request validation.
  - `Responsibilities:` Receives the request from the client and validates it.
- `close()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Closes the connection to the client.

3.5 `AsyncRequestIterator` Class

The `AsyncRequestIterator` class is responsible for asynchronously iterating over incoming HTTP requests and passing them to the appropriate handler based on the route and method.

Initialization

- `Inputs:`
  - `router` (Router): An instance of the `Router` class to handle request routing.
- `Outputs:` None
- `Responsibilities:` Initializes the asynchronous request iterator with the specified router.

Attributes

- `router`: An instance of the `Router` class that handles request routing.

Methods

- `__aiter__()`:
  - `Inputs:` None
  - `Outputs:`
    - `self` (`AsyncRequestIterator`): Returns the asynchronous iterator instance.
  - `Responsibilities:` Returns the asynchronous iterator instance.
- `__anext__()`:
  - `Inputs:` None
  - `Outputs:`
    - `response` (str): The response generated after handling the request.
  - `Responsibilities:` Asynchronously retrieves the next request from the router, processes it, and returns the response. Raises `StopAsyncIteration` when there are no more requests to process.
- `pass_to_handler(request, routes, http_request_line, headers, body)`:
  - `Inputs:`
    - `request` (str): The original raw HTTP request text.
    - `routes` (dict): A dictionary of registered routes.
    - `http_request_line` (dict): Parsed HTTP request line containing method, path, and HTTP version.
    - `headers` (dict): Parsed HTTP headers.
    - `body` (dict or str): Parsed HTTP body.
  - `Outputs:`
    - `response` (str): The response generated after handling the request.
  - `Responsibilities:` Determines the appropriate request handler based on the HTTP method and route, processes the request, and returns the response.

3.6 `BaseRequestHandler` Class

The `BaseRequestHandler` class is an abstract base class for handling HTTP requests. It provides common functionality for parsing and validating requests, and is extended by specific request handler classes.

Initialization

- `Inputs:`
  - `request_text` (str): The raw HTTP request text.
  - `needs_auth` (bool): Indicates whether the request requires authentication. Default is `False`.
- `Outputs:` None
- `Responsibilities:` Initializes the request handler with the specified request text and authentication requirement. Parses the request text into its components.

Attributes

- `request_text`: The raw HTTP request text.
- `http_request_line`: A dictionary containing the HTTP request line components (method, path, HTTP version).
- `headers`: A dictionary of HTTP headers.
- `body`: The HTTP request body.
- `needs_auth`: Indicates whether the request requires authentication.

Methods

- `parse_request_text()`:
  - `Inputs:` None
  - `Outputs:`
    - `http_request_line` (dict): Parsed HTTP request line containing method, path, and HTTP version.
    - `headers` (dict): Parsed HTTP headers.
    - `body` (dict or str): Parsed HTTP body.
  - `Responsibilities:` Parses the raw HTTP request text into its components.
- `format_request()`:
  - `Inputs:` None
  - `Outputs:`
    - `request` (str): The formatted HTTP request text.
  - `Responsibilities:` Formats the parsed request components back into a raw HTTP request text.
- `__repr__()`:
  - `Inputs:` None
  - `Outputs:`
    - `request_text` (str): The raw HTTP request text.
  - `Responsibilities:` Returns the raw HTTP request text representation of the object.
- `validate_request()`:
  - `Inputs:` None
  - `Outputs:`
    - `is_valid` (bool): Indicates whether the request is valid.

    * `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the HTTP request, including the request line, method, HTTP version, headers, body, and authorization if required.

- `validate_request_line(lines)`:
  - `Inputs:`
    * `lines` (list): List of lines from the raw HTTP request text.
  - `Outputs:`
    * `is_valid` (bool): Indicates whether the request line is valid.
    * `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the HTTP request line format.

- `validate_authorization()`:
  - `Inputs:` None
  - `Outputs:`
    * `is_valid` (bool): Indicates whether the authorization is valid.
    * `message` (str): A message indicating the result of the authorization validation.
  - `Responsibilities:` Validates the authorization header if the request requires authentication.

- `validate_method(method)`:
  - `Inputs:`
    * `method` (str): The HTTP method.
  - `Outputs:`
    * `is_valid` (bool): Indicates whether the method is valid.
    * `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the HTTP method (GET or POST).

- `validate_http_version(http_version)`:
  - `Inputs:`
    * `http_version` (str): The HTTP version.
  - `Outputs:`
    * `is_valid` (bool): Indicates whether the HTTP version is valid.
    * `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the HTTP version.

- `parse_headers_and_body(lines)`:
  - `Inputs:`
    * `lines` (list): List of lines from the raw HTTP request text.
  - `Outputs:`
    * `headers` (dict): Parsed HTTP headers.
    * `body_lines` (list): List of lines from the HTTP body.
    * `is_valid` (bool): Indicates whether the headers and body are valid.
    * `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Parses the HTTP headers and body from the request text.

- `validate_post_request(headers, body_lines)`:

  - `Inputs:`
    - `headers` (dict): Parsed HTTP headers.
    - `body_lines` (list): List of lines from the HTTP body.
  - `Outputs:`
    - `is_valid` (bool): Indicates whether the POST request is valid.
    - `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the headers and body of a POST request.

- `validate_get_request(body_lines)`:
  - `Inputs:`
    - `body_lines` (list): List of lines from the HTTP body.
  - `Outputs:`
    - `is_valid` (bool): Indicates whether the GET request is valid.
    - `message` (str): A message indicating the result of the validation.
  - `Responsibilities:` Validates the body of a GET request.

- `handle_request()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` An abstract method to be implemented by subclasses for handling the request.

3.7 `GetRequestHandler` Class

The `GetRequestHandler` class extends `BaseRequestHandler` and handles HTTP GET requests.

Initialization

- `Inputs:`
  - `request_text` (str): The raw HTTP request text. Default is `None`.
  - `needs_auth` (bool): Indicates whether the request requires authentication. Default is `False`.
- `Outputs:` None
- `Responsibilities:` Initializes the GET request handler with the specified request text and authentication requirement.

Methods

- `handle_request(func)`:
  - `Inputs:`
    - `func` (function): The function to handle the GET request.
  - `Outputs:`
    - `response` (str): The response generated by the function.
  - `Responsibilities:` Asynchronously handles the GET request and returns the response generated by the function.

3.8 `PostRequestHandler` Class

The `PostRequestHandler` class extends `BaseRequestHandler` and handles HTTP POST requests.

Initialization

- `Inputs:`
  - `request_text` (str): The raw HTTP request text. Default is `None`.
  - `needs_auth` (bool): Indicates whether the request requires authentication. Default is `False`.
- `Outputs:` None
- `Responsibilities:` Initializes the POST request handler with the specified request text and authentication requirement.

Methods

- `handle_request(func)`:
  - `Inputs:`
    - `func` (function): The function to handle the POST request.
  - `Outputs:`
    - `response` (str): The response generated by the function.
  - `Responsibilities:` Asynchronously handles the POST request and returns the response generated by the function.

3.9 `ResponseHandler` Class

The `ResponseHandler` class is responsible for formatting HTTP responses. It encapsulates the HTTP version, status code, headers, and body of the response.

Initialization

- `Inputs:`
  - `http_version` (str): The HTTP version of the response (e.g., `"HTTP/1.1"`).
  - `status_code` (str): The HTTP status code of the response (e.g., `"200 OK"`).
  - `headers` (dict): A dictionary of HTTP headers.
  - `body` (str): The body of the response.
- `Outputs:` None
- `Responsibilities:` Initializes the response handler with the specified HTTP version, status code, headers, and body. Raises a `ValueError` if any parameter is `None`.

Attributes

- `http_version`: The HTTP version of the response.
- `status_code`: The HTTP status code of the response.
- `headers`: A dictionary of HTTP headers.
- `body`: The body of the response.

Methods

- `format_response()`:
  - `Inputs:` None
  - `Outputs:`
    - `response` (str): The formatted HTTP response text.
  - `Responsibilities:` Formats the HTTP version, status code, headers, and body into a raw HTTP response text.
- `__repr__()`:
  - `Inputs:` None
  - `Outputs:`
    - `response_text` (str): The formatted HTTP response text.
  - `Responsibilities:` Returns the raw HTTP response text representation of the object.

3.10 Decorators

The decorators provided in this module add additional functionality to the request handling process, including logging requests and authorizing them.

3.10.1 `log_requests` Decorator

The `log_requests` decorator logs the details of each request processed by the decorated function.

Usage

- `Inputs:`
  - `func` (function): The function to be decorated.
- `Outputs:`
  - `logger_wrapper` (function): The decorated function that logs request details.
- `Responsibilities:` Logs the start of request logging, the details of the request, and a separator line.

3.10.2 `authorize_request` Decorator

The `authorize_request` decorator checks the authorization of the request processed by the decorated function.

Usage

- `Inputs:`
  - `func` (function): The function to be decorated.
- `Outputs:`
  - `check_auth` (function): The decorated function that checks request authorization.
- `Responsibilities:`
  - Checks if the request is a POST or GET request.
  - Validates the request using the appropriate request handler.
  - Checks if the route exists and if it matches the method of the request.
  - Validates the authorization if the route requires authentication.
  - Returns the request, validation status, and validation message.

3.11 Generators

The generators provided in this module are responsible for handling the streaming of responses.

3.11.1 `streaming_response_generator` Function

The `streaming_response_generator` function is responsible for streaming the HTTP response in chunks.

Usage

- `Inputs:`
  - `response` (ResponseHandler): An instance of the `ResponseHandler` class containing the response details (HTTP version, status code, headers, and body).
- `Outputs:`
  - `chunk` (str): A chunk of the HTTP response to be streamed.
- `Responsibilities:`
  - Constructs the status line and headers from the response.
  - Splits the response body into chunks.
  - Yields each chunk of the response.

3.12 Context Managers

The context manager provided in this module helps in managing the lifecycle of the web server, ensuring it starts and stops appropriately.

3.12.1 `ServerContextManager` Class

The `ServerContextManager` class is responsible for managing the web server using context management protocols.

Initialization

- `Inputs:`
  - `webserver` (WebServer): An instance of the `WebServer` class to be managed.
- `Outputs:` None
- `Responsibilities:` Initializes the context manager with the specified web server instance.

Methods

- `__enter__()`:
  - `Inputs:` None
  - `Outputs:` None
  - `Responsibilities:` Starts the web server by calling its `listen` method.
- `__exit__()`:
  - `Inputs:`
    - `exc_type` (type): The exception type (if any) that caused the context to be exited.
    - `exc_value` (Exception): The exception instance (if any) that caused the context to be exited.

    * `exc_traceback` (traceback): The traceback (if any) of the exception that caused the context to be exited.

- `Outputs:` None
- `Responsibilities:` Stops the web server by calling its `close` method, ensuring that the server is properly shut down even if an exception occurs.