Web Server

By

Maha Abdullah Alassaf

Overview

This project aims to create a robust web server in Python capable of handling HTTP requests (GET and POST) and implementing advanced features such as decorators, generators, iterators, async handling, context managers, and the singleton pattern.

Table of Contents

Overview

This project aims to create a robust web server in Python capable of handling HTTP requests (GET and POST) and implementing advanced features such as decorators, generators, iterators, async handling, context managers, and the singleton pattern.

Chapter 1 - Overview

1.1 Problem Statement & Significance

This project aims to create a robust web server in Python capable of handling HTTP requests (GET and POST) and implementing advanced features such as decorators, generators, iterators, async handling, context managers, and the singleton pattern.

1.2 Definitions of New Termsm

- **Decorators**: Functions that modify the behavior of other functions.
- **Generators**: Functions that enable streaming of data in chunks.
- **Iterators**: Objects that enable looping over a sequence of data.
- **Async Handling**: Techniques for processing requests asynchronously.
- **Context Managers**: Objects used to manage resources within a specific scope.
- **Singleton Pattern**: Design pattern ensuring a single instance of a class exists.

1.3 Analysis of Project Requirements

The goal of this project is to develop a robust web server capable of handling GET and POST requests while leveraging advanced Python features to ensure efficient, scalable, and maintainable code. The key requirements and features needed are:

features	Objective	Implementation
Handling HTTP Requests (GET and POST):	Manage different types of client requests and respond appropriately	Create handlers for GET and POST requests to process and respond to client interactions.
Decorators for Logging and Authorization:	Log requests for debugging and monitoring purposes, and ensure that only authorized requests are processed.	Use the log_request decorator to log details of each request and the authorize_request decorator to check and authorize requests.
Generators for Streaming Responses:	Efficiently handle large responses by sending data incrementally rather than all at once.	Implement streaming_response_generator to yield parts of the response, ensuring the server can handle large payloads without consuming too much memory.
Iterators to Manage Multiple Requests:	Sequentially process multiple incoming requests in a controlled manner.	Create the RequestIterator class to manage and iterate over multiple requests.
Coroutines and Async Iterators for Asynchronous Request Handling:	Improve the server's responsiveness and scalability by handling multiple requests asynchronously.	Use the async_request_handler function to process requests asynchronously with async for and RequestIterator.
Inheritance and Polymorphism for Request Handlers:	Create a modular and extensible structure for handling different types of requests.	Define a base class BaseRequestHandler and derive GetRequestHandler and PostRequestHandler classes to handle GET and POST requests respectively.
Context Managers for Server Lifecycle Management:	Ensure proper resource management during the server's lifecycle (start and stop).	Use ServerContextManager to manage the initialization and shutdown of the server.
Singleton Pattern to Ensure a Single Server Instance:	Prevent multiple instances of the server from running simultaneously, which could cause conflicts.	mplement the WebServer class with the singleton pattern to guarantee only one server instance is active.
Streaming Responses for Incremental Data Sending:	Enhance the server's performance by sending data in chunks.	Modify response handlers to utilize streaming_response_generator for incremental data transmission.

1.4 Features Implemented Solution

This project addresses the need for a scalable and efficient web server capable of handling diverse client requests. Key objectives include:

- **Handling HTTP Requests**: Manage GET and POST requests effectively.
- **Logging and Authorization**: Ensure requests are logged for monitoring and only authorized requests are processed.
- **Streaming Responses**: Efficiently transmit large data payloads using generators.
- **Request Management**: Sequentially process multiple requests using iterators.
- **Asynchronous Handling**: Improve server responsiveness with async iterators.
- **Lifecycle Management**: Ensure proper server start-up and shutdown using context managers.
- **Singleton Pattern**: Maintain a single instance of the server to prevent conflicts.

1.5 Project structure

- webserver.py: Main file containing the complete server implementation.
- Decorators fun: Contains decorators for logging and authorization.
- handlers.py fu: Defines request handler classes for GET, POST, and shutdown requests.
- generators.py fun: Implements generators for streaming responses.
- Iterators fun: Manages multiple requests using iterators.
- async_handlers fun: Includes async handlers for asynchronous request processing.
- context_managers fun: Defines context managers for server lifecycle management.
- Singleton fun: Implements the singleton pattern for ensuring a single server instance

Chapter 2 – Code

2.1 Simple Web Server

The project starts with a simple HTTP server setup using Python's built-in `http.server` module. This serves as a foundational example before integrating advanced features.

```
from http.server import BaseHTTPRequestHandler, HTTPServer
```

```
# Simple request handler
class SimpleRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(b"<html><body><h1>Simple GET Request</h1></body></html>")

# Function to start the simple server
def start_simple_server():
    server_address = ('', 9000)
    httpd = HTTPServer(server_address, SimpleRequestHandler)
    print("Starting simple server on port 9000...")
    httpd.serve_forever()
```

2.2 Implementing Decorators for Logging and Authorization

- **log_request decorator**: details of each request, including function name, client address, and request path.
- **authorize_request decorator:** Authorizes requests; if unauthorized, sends a 403 Forbidden response.

```
# Implementing Decorators
def log_request(func):
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        print(f"\nFunction {func.__name__} called")
        print(f"Request from: {self.client_address}")
        print(f"Request path: {self.path}")
        result = func(self, *args, **kwargs)
        print(f"Function {func.__name__} executed\n")
        return result
    return wrapper

def authorize_request(func):
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        print("Authorizing request...")
        authorized = True
        if authorized:
            print("Request authorized")
            return func(self, *args, **kwargs)
        else:
            self.send_response(403)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
```

```
                   self.wfile.write(json.dumps({"error": "403 Forbidden", "message": "You are
not authorized to access this page."}).encode())
                   print("Unauthorized request")
                   return
           return wrapper
```

2.3 Implementing the Request Handler Class

The project implements request handler classes to manage different types of HTTP requests (`GET`, `POST`, and shutdown).

- **BaseRequestHandler**: Abstract base class defining the structure for handling requests.
- **GetRequestHandler**: Handles `GET` requests by responding with a JSON payload.
- **PostRequestHandler**: Handles `POST` requests by processing incoming data and responding with a JSON payload.
- **ShutdownRequestHandler**: Handles server shutdown requests gracefully.

```
class BaseRequestHandler(ABC):
    @abstractmethod
    def handle_request(self, handler):
        pass

class GetRequestHandler(BaseRequestHandler):
    def handle_request(self, handler):
        client_info = {
            "client_ip": handler.client_address[0],
            "client_port": handler.client_address[1],
            "user_agent": handler.headers.get("User-Agent"),
            "content_type": handler.headers.get("Content-Type")
        }
        print(f"Handling GET request with client info: {client_info}")

        response_data = {
            "status": "success",
            "method": "GET",
            "path": handler.path,
            "message": "GET request response",
            "client_info": client_info
        }

        response_json = json.dumps(response_data, indent=4)
        handler.send_response(200)
        handler.send_header('Content-Type', 'application/json')
        handler.end_headers()

        try:
            start_time = time.time()
            for part in streaming_response_generator(response_json):
                handler.wfile.write(part.encode())
            end_time = time.time()
            print(f"Streaming response time: {end_time - start_time:.4f} seconds")
        except (ConnectionResetError, BrokenPipeError):
            print("Connection lost while sending response")

class PostRequestHandler(BaseRequestHandler):
    def handle_request(self, handler):
        client_info = {
            "client_ip": handler.client_address[0],
            "client_port": handler.client_address[1],
            "user_agent": handler.headers.get("User-Agent"),
            "content_type": handler.headers.get("Content-Type")
        }
        print(f"Handling POST request with client info: {client_info}")

        content_length = int(handler.headers['Content-Length'])
        post_data = handler.rfile.read(content_length)
        post_data = urllib.parse.parse_qs(post_data.decode('utf-8'))
        new_task = post_data.get('task', [''])[0]

        print(f"Received data: {post_data}")

        response_data = {
            "status": "success",
            "method": "POST",
            "path": handler.path,
            "message": f"POST request data: {post_data}",
            "client_info": client_info
        }

        response_json = json.dumps(response_data, indent=4)
        handler.send_response(200)
        handler.send_header('Content-Type', 'application/json')
        handler.end_headers()

        try:
            start_time = time.time()
            for part in streaming_response_generator(response_json):
                handler.wfile.write(part.encode())
            end_time = time.time()
            print(f"Streaming response time: {end_time - start_time:.4f} seconds")
        except (ConnectionResetError, BrokenPipeError):
            print("Connection lost while sending response")

class ShutdownRequestHandler(BaseRequestHandler):
    def handle_request(self, handler):
        print("Handling server shutdown request")
        handler.send_response(200)
        handler.send_header('Content-Type', 'application/json')
        handler.end_headers()
        handler.wfile.write(json.dumps({"message": "Server is shutting down..."}).encode())
        threading.Thread(target=handler.server.shutdown).start()
```

2.3.1 RequestHandler Class

```
class RequestHandler(BaseHTTPRequestHandler):
    @log_request
    @authorize_request
    def do_GET(self):
        print("Function do_GET called")
        handler = GetRequestHandler()
        handler.handle_request(self)
        print("Function do_GET executed")

    @log_request
    @authorize_request
    def do_POST(self):
        if self.path == '/shutdown':
            handler = ShutdownRequestHandler()
        else:
            handler = PostRequestHandler()
        print("Function do_POST called")
        handler.handle_request(self)
        print("Function do_POST executed")
```

2.4 Implementing Generators for Streaming Responses

2.4.1 Generators

```
def streaming_response_generator(message):
    chunk_size = 50
    total_chunks = (len(message) + chunk_size - 1) // chunk_size
    print(f"Total message length: {len(message)} characters")
    for i in range(0, len(message), chunk_size):
        chunk = message[i:i + chunk_size]
        print(f"Streaming chunk: {chunk}")
        yield chunk
    print("Completed streaming all chunks.")
```

2.4.2 Iterator

```
class RequestIterator:
    def __init__(self, requests):
        self._requests = requests
        self._index = 0
        print("Initialized RequestIterator with requests:", requests)
```

```python
    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._requests):
            result = self._requests[self._index]
            print(f"RequestIterator returning request at index {self._index}: {result}")
            self._index += 1
            return result
        else:
            print("RequestIterator reached end of requests")
            raise StopIteration

    def __len__(self):
        return len(self._requests)

# Implementing Async Iterator
class AsyncRequestIterator:
    def __init__(self, request_iterator):
        self._request_iterator = request_iterator
        self._requests = list(request_iterator)
        self._index = 0
        print("Initialized AsyncRequestIterator with requests:", self._requests)

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self._index < len(self._requests):
            result = self._requests[self._index]
            print(f"AsyncRequestIterator returning request at index {self._index}: {result}")
            self._index += 1
            await asyncio.sleep(0)  # Simulate async processing time
            return result
        else:
            print("AsyncRequestIterator reached end of requests")
            raise StopAsyncIteration

async def async_request_handler(request_iterator):
    print("async_request_handler called")
    async for request in AsyncRequestIterator(request_iterator):
        print(f"Processing request asynchronously: {request}")
        await asyncio.sleep(1)
    print("async_request_handler executed")
```

2.4.3 Singleton Pattern

```python
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            instance = super(SingletonMeta, cls).__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

class WebServer(metaclass=SingletonMeta):
    def __init__(self, server_address, handler_class):
        self.server = HTTPServer(server_address, handler_class)
```

2.4.4 Context Manager

```
class ServerContextManager:
    def __init__(self, server_address, handler_class):
        self.server_instance = WebServer(server_address, handler_class)

    def __enter__(self):
        start_time = time.time()
        print("Initializing ServerContextManager")
        end_time = time.time()
        print(f"ServerContextManager initialized in {end_time - start_time:.4f} seconds")
        print(f"Starting server on {self.server_instance.server.server_address}...")
        return self.server_instance.server

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Shutting down server")
        self.server_instance.server.shutdown()
        self.server_instance.server.server_close()
```

2.5 Main

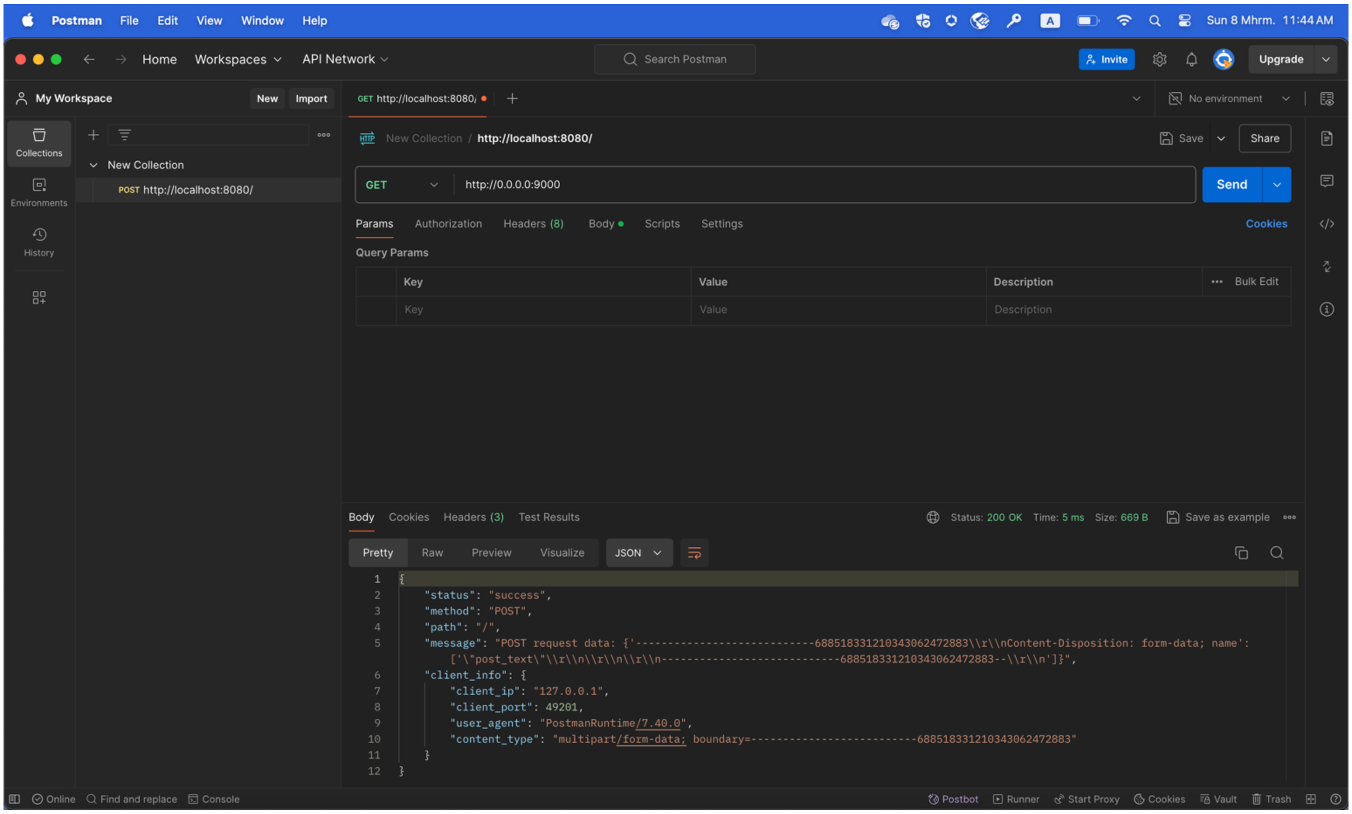
```
def run():
    server_address = ('0.0.0.0', 9000)
    handler_class = RequestHandler
    print("Setting up server: Initializing HTTP server and handler class")

    with ServerContextManager(server_address, handler_class) as httpd:
        print("*************** Welcome to Maha's Server ***************")
        print(f"---------\nServer running at
http://{server_address[0]}:{server_address[1]}\n")

        try:
            httpd.serve_forever()
        except KeyboardInterrupt:
            print("\nShutting down server...")

if __name__ == "__main__":
```

```
run()
```

Chapter 3 – Testing

3.1 Get and Post

3.1.1 Get request

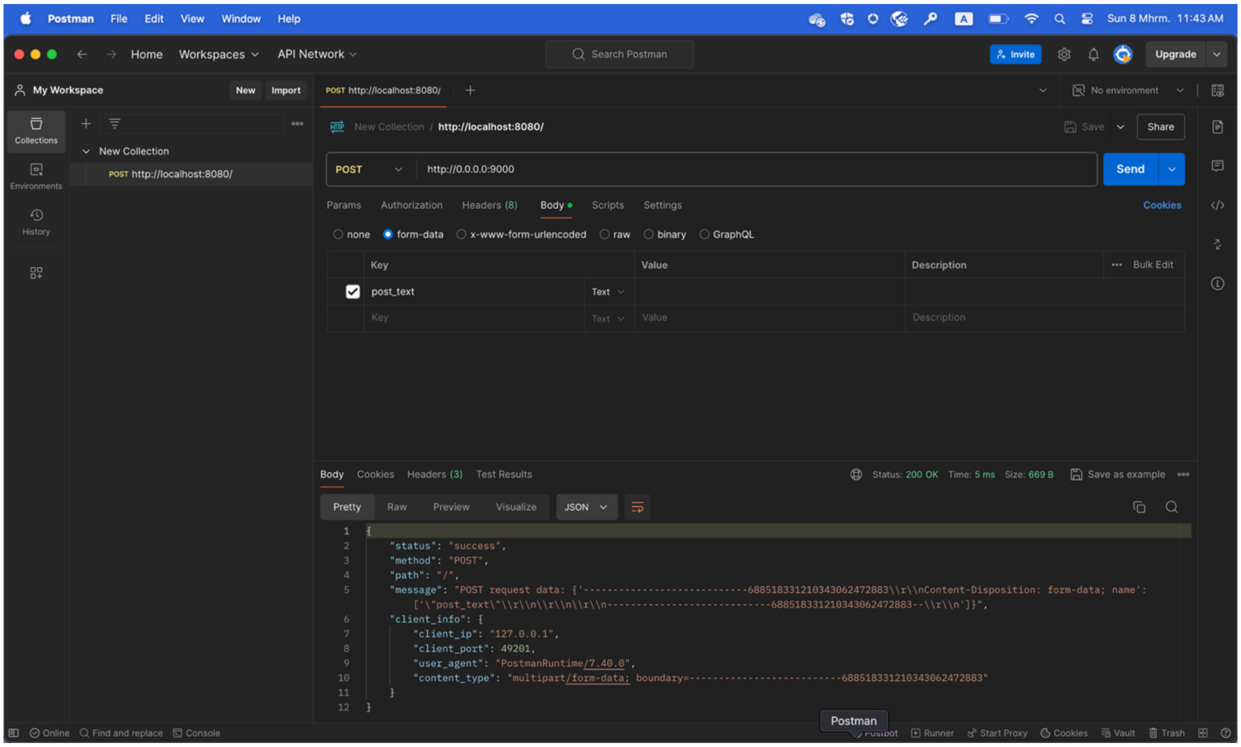


GET http://0.0.0.0:9000

Status: 200 OK Time: 5 ms Size: 669 B

```
{
    "status": "success",
    "method": "POST",
    "path": "/",
    "message": "POST request data: {'----------------------------688518331210343062472883\\r\\nContent-Disposition: form-data; name':
        ['\"post_text\"\\r\\n\\r\\n\\r\\n----------------------------688518331210343062472883--\\r\\n']}",
    "client_info": {
        "client_ip": "127.0.0.1",
        "client_port": 49201,
        "user_agent": "PostmanRuntime/7.40.0",
        "content_type": "multipart/form-data; boundary=--------------------------688518331210343062472883"
    }
}
```

3.1.2 Post request


Postman File Edit View Window Help
Sun 8 Mhrm. 11:43 AM
Home Workspaces API Network
Search Postman
Invite Upgrade
My Workspace New Import
Collections Environments History
New Collection
POST http://localhost:8080/
No environment
New Collection / http://localhost:8080/
Save Share
Send
Params Authorization Headers (8) Body Scripts Settings Cookies
Body Cookies Headers (3) Test Results
Pretty Raw Preview Visualize JSON
Save as example
Online Find and replace Console
Postman
Postbot Runner Start Proxy Cookies Vault Trash

POST http://0.0.0.0:9000

○ none ● form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

	Key		Value	Description
☑	post_text	Text		
	Key	Text	Value	Description

Status: 200 OK Time: 5 ms Size: 669 B

```
{
    "status": "success",
    "method": "POST",
    "path": "/",
    "message": "POST request data: {'----------------------------688518331210343062472883\\r\\nContent-Disposition: form-data; name':
        ['\"post_text\"\\r\\n\\r\\n\\r\\n----------------------------688518331210343062472883--\\r\\n']}",
    "client_info": {
        "client_ip": "127.0.0.1",
        "client_port": 49201,
        "user_agent": "PostmanRuntime/7.40.0",
        "content_type": "multipart/form-data; boundary=--------------------------688518331210343062472883"
    }
}
```

3.2 Unit test

The unit tests are provided to ensure the correctness and robustness of the server's features. The tests cover decorators, generators, iterators, asynchronous handling, and the singleton pattern.

```python
import unittest
from unittest.mock import MagicMock, patch
from http.server import BaseHTTPRequestHandler
import asyncio
from iterators import RequestHandler
from webserver import (log_request, authorize_request, GetRequestHandler, PostRequestHandler,
                       ShutdownRequestHandler, streaming_response_generator, RequestIterator,
                       AsyncRequestIterator, async_request_handler, WebServer,
                       ServerContextManager)

# Test Decorators
class TestDecorators(unittest.TestCase):
    def test_log_request(self):
        print("Running TestDecorators.test_log_request")
        @log_request
        def mock_func(self):
            return "called"

        mock_handler = MagicMock()
        mock_handler.client_address = ('127.0.0.1', 8080)
        mock_handler.path = '/'

        result = mock_func(mock_handler)
        self.assertEqual(result, "called")
        print("TestDecorators.test_log_request passed\n")

    def test_authorize_request(self):
        print("Running TestDecorators.test_authorize_request")
        @authorize_request
        def mock_func(self):
            return "authorized"

        mock_handler = MagicMock()
        mock_handler.client_address = ('127.0.0.1', 8080)
        mock_handler.path = '/'

        result = mock_func(mock_handler)
        self.assertEqual(result, "authorized")
        print("TestDecorators.test_authorize_request passed\n")

# Test Generators
class TestGenerators(unittest.TestCase):
    def test_streaming_response_generator(self):
        print("Running TestGenerators.test_streaming_response_generator")
        message = "This is a test message."
        chunks = list(streaming_response_generator(message))
        self.assertEqual("".join(chunks), message)
        print("TestGenerators.test_streaming_response_generator passed\n")

# Test Iterators
class TestRequestIterator(unittest.TestCase):
    def test_request_iterator(self):
        print("Running TestRequestIterator.test_request_iterator")
        requests = ["request1", "request2", "request3"]
        iterator = RequestIterator(requests)
        self.assertEqual(len(iterator), 3)
        self.assertEqual(list(iterator), requests)
        print("TestRequestIterator.test_request_iterator passed\n")

class TestAsyncRequestIterator(unittest.IsolatedAsyncioTestCase):
    async def test_async_request_iterator(self):
        print("Running TestAsyncRequestIterator.test_async_request_iterator")
        requests = ["request1", "request2", "request3"]
        iterator = AsyncRequestIterator(RequestIterator(requests))
        result = [req async for req in iterator]
        self.assertEqual(result, requests)
        print("TestAsyncRequestIterator.test_async_request_iterator passed\n")

class TestAsyncRequestHandler(unittest.IsolatedAsyncioTestCase):
    async def test_async_request_handler(self):
        print("Running TestAsyncRequestHandler.test_async_request_handler")
        requests = ["request1", "request2", "request3"]
        iterator = RequestIterator(requests)

        await async_request_handler(iterator)
        print("TestAsyncRequestHandler.test_async_request_handler passed\n")

class TestGetRequestHandler(unittest.TestCase):
    def setUp(self):
        self.handler = GetRequestHandler()

    def test_handle_request(self):
        print("Running TestGetRequestHandler.test_handle_request")
        mock_handler = MagicMock(spec=BaseHTTPRequestHandler)
        mock_handler.client_address = ('127.0.0.1', 8080)
        mock_handler.headers = {
            'User-Agent': 'TestAgent',
            'Content-Type': 'application/json'
        }
        mock_handler.path = '/'
        mock_handler.wfile = MagicMock()

        self.handler.handle_request(mock_handler)

        mock_handler.wfile.write.assert_called()
        print("TestGetRequestHandler.test_handle_request passed\n")

class TestPostRequestHandler(unittest.TestCase):
    def setUp(self):
        self.handler = PostRequestHandler()

    def test_handle_request(self):
        print("Running TestPostRequestHandler.test_handle_request")
        mock_handler = MagicMock(spec=BaseHTTPRequestHandler)
        mock_handler.client_address = ('127.0.0.1', 8080)
        mock_handler.headers = {
            'User-Agent': 'TestAgent',
            'Content-Length': '50',
            'Content-Type': 'application/json'
        }
        mock_handler.rfile = MagicMock()
        mock_handler.rfile.read = MagicMock(return_value=b'task=sample_task')
        mock_handler.path = '/'
        mock_handler.wfile = MagicMock()
        self.handler.handle_request(mock_handler)

        mock_handler.wfile.write.assert_called()
        print("TestPostRequestHandler.test_handle_request passed\n")

class TestShutdownRequestHandler(unittest.TestCase):
    def setUp(self):
        self.handler = ShutdownRequestHandler()

    def test_handle_request(self):
        print("Running TestShutdownRequestHandler.test_handle_request")
        mock_handler = MagicMock(spec=BaseHTTPRequestHandler)
        mock_handler.server = MagicMock()
        mock_handler.wfile = MagicMock()

        self.handler.handle_request(mock_handler)

        mock_handler.server.shutdown.assert_called_once()
        print("TestShutdownRequestHandler.test_handle_request passed\n")

class TestWebServer(unittest.TestCase):
    def test_singleton(self):
        print("Running TestWebServer.test_singleton")
        server1 = WebServer(('0.0.0.0', 9000), RequestHandler)
        server2 = WebServer(('0.0.0.0', 9000), RequestHandler)
        self.assertIs(server1, server2)
        print("TestWebServer.test_singleton passed\n")

class TestServerContextManager(unittest.TestCase):
    def test_context_manager(self):
        print("Running TestServerContextManager.test_context_manager")
        with patch('webserver.HTTPServer') as MockHTTPServer:
            MockHTTPServer.return_value = MagicMock()
            with ServerContextManager(('0.0.0.0', 9000), RequestHandler) as server:
                self.assertIsInstance(server, MagicMock)
        print("TestServerContextManager.test_context_manager passed\n")

if __name__ == '__main__':
    unittest.main()
```