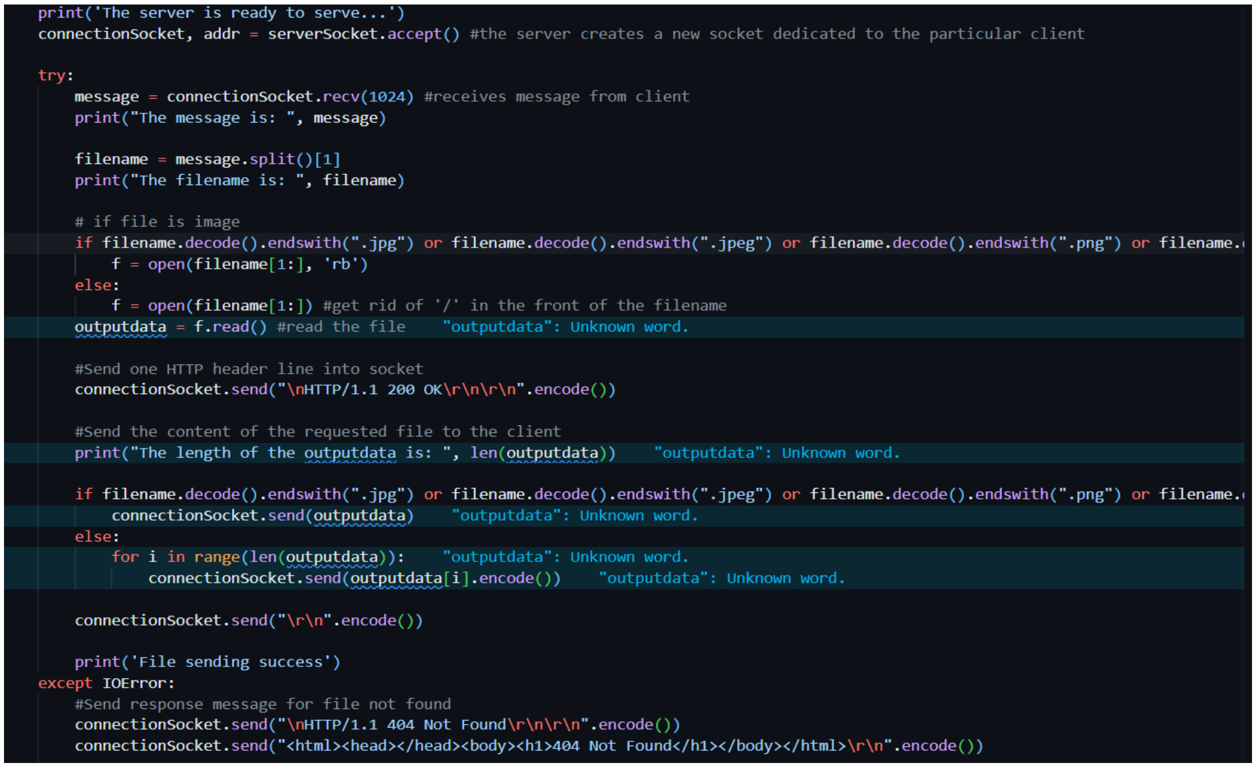
This project involves the development of a simple web server using Python's socket programming. The web server is designed to handle HTTP GET requests from clients and respond with requested files, including HTML pages and image files. The server listens for incoming connections, processes requests, and serves files accordingly.

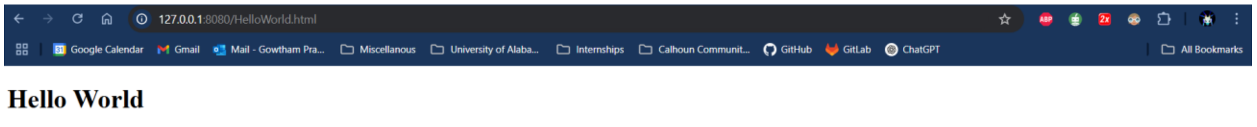
The server is implemented using the socket module in Python. It establishes a TCP connection to listen for incoming client requests and process them. The key details of the server include the server IP address, which is localhost (127.0.0.1), and the port number, which is 8080. The server uses the TCP protocol and supports HTTP/1.1. The server is designed to accept connections and handle requests in a loop, ensuring continuous service unless manually stopped.

The server follows a structured approach in handling requests. It begins by listening for requests, being bound to localhost on port 8080. When a client sends a request, the server accepts the connection and creates a new socket dedicated to that client. The request is then processed by reading the HTTP GET request and extracting the filename from the request URL. If the requested file is an HTML document, it is read as a text file and sent to the client. If the requested file is an image, such as JPEG, PNG, or GIF, it is read in binary mode and sent accordingly. If the requested file is not found, the server responds with an HTTP 404 Not Found message. Once the file is served, the client connection is closed, and the server waits for the next request. A code snippet is shown below.



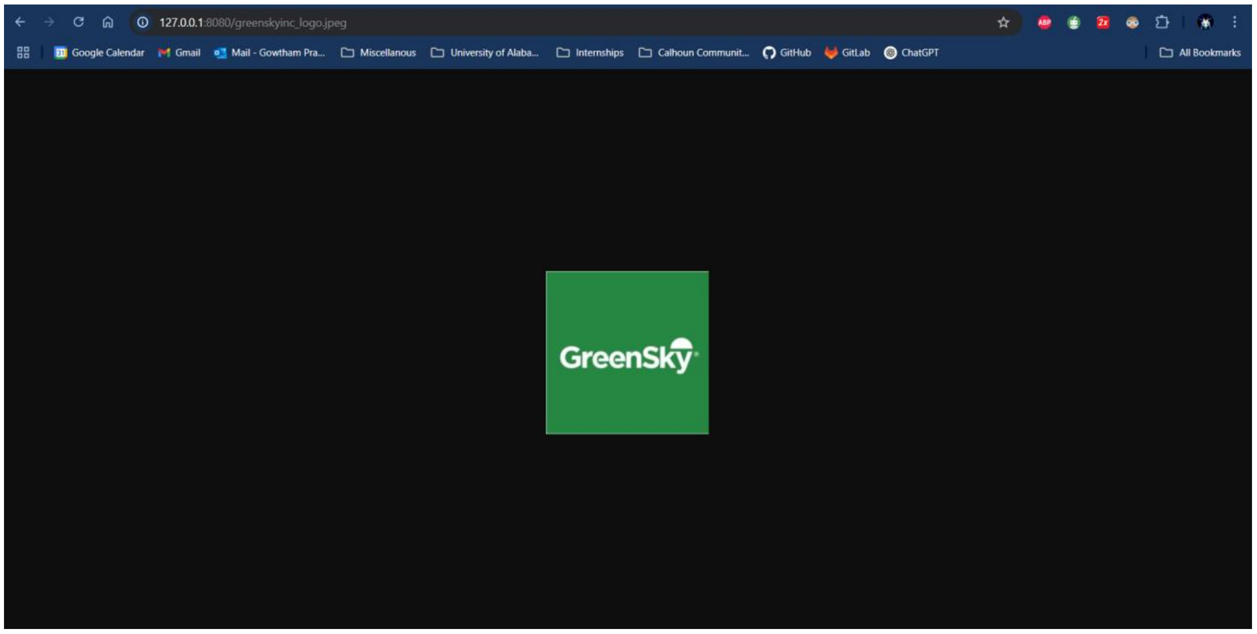
```
print('The server is ready to serve...')
connectionSocket, addr = serverSocket.accept() #the server creates a new socket dedicated to the particular client

try:
    message = connectionSocket.recv(1024) #receives message from client
    print("The message is: ", message)

    filename = message.split()[1]
    print("The filename is: ", filename)

    # if file is image
    if filename.decode().endswith(".jpg") or filename.decode().endswith(".jpeg") or filename.decode().endswith(".png") or filename.
        f = open(filename[1:], 'rb')
    else:
        f = open(filename[1:]) #get rid of '/' in the front of the filename
    outputdata = f.read() #read the file    "outputdata": Unknown word.

    #Send one HTTP header line into socket
    connectionSocket.send("\nHTTP/1.1 200 OK\r\n\r\n".encode())

    #Send the content of the requested file to the client
    print("The length of the outputdata is: ", len(outputdata))    "outputdata": Unknown word.

    if filename.decode().endswith(".jpg") or filename.decode().endswith(".jpeg") or filename.decode().endswith(".png") or filename.
        connectionSocket.send(outputdata)    "outputdata": Unknown word.
    else:
        for i in range(len(outputdata)):    "outputdata": Unknown word.
            connectionSocket.send(outputdata[i].encode())    "outputdata": Unknown word.

    connectionSocket.send("\r\n".encode())

    print('File sending success')
except IOError:
    #Send response message for file not found
    connectionSocket.send("\nHTTP/1.1 404 Not Found\r\n\r\n".encode())
    connectionSocket.send("<html><head></head><body><h1>404 Not Found</h1></body></html>\r\n".encode())
```

To ensure the functionality of the web server, two test cases were executed. The first test involved requesting an HTML file named HelloWorld.html. A simple HTML file was created and placed in the server directory. When requested through the browser or a client script, the server successfully located and served the file, displaying it correctly in the browser. The expected outcome was that the browser would render the contents of HelloWorld.html, which was achieved successfully and is shown below.



Hello World

The second test involved requesting an image file named test_image.jpeg. A JPEG image was placed in the server directory, and when requested, the server read the file in binary mode and sent it correctly to the client. The expected outcome was that the image would display properly in the browser, which was confirmed during testing and is shown below.



The server includes error handling mechanisms to deal with potential issues. If a client requests a non-existent file, the server responds with a 404 Not Found error and an HTML error page. Connection errors are managed gracefully, ensuring stability. Additionally, the server can be manually stopped using CTRL+C, ensuring proper closure of sockets before termination.

In conclusion, this project successfully demonstrates the implementation of a basic web server using Python socket programming. The server can handle client requests, serve both HTML and image files, and manage errors effectively. The test cases confirmed its functionality, proving its ability to serve different types of files reliably. Future improvements may include support for additional HTTP methods such as POST and PUT, multi-threading for handling multiple clients simultaneously, and enhanced security features.