

Design Document

Introduction:

We used Java to implement basic HTTP server. Our implementation design used an HTTP server API named “com.sun.net.httpserver”.

In our design, We created a `HttpServer` object with port number 8000. After that we created contexts according to our need. And then we started the server.

A context describes the pattern in request url for a matching Handler. A Handler is responsible for further processing of the request. For that a Handler should implement a `HttpHandler` with overriding ‘`handle(HttpExchange he)`’ method.

This `HttpExchange` class encapsulates a HTTP request received and a response to be generated in one exchange. It provides methods for examining the request from the client, and for building and sending the response[\[1\]](#)

The following headers have been used in our design:

1. `getResponseHeaders()` to set any response headers
2. `sendResponseHeaders(int,long)` to send the response headers.
3. `getResponseBody()` to get a `OutputStream` to send the response body. When the response body has been written, the stream is closed to terminate the exchange.

The features of our server are:

- it can handle multiple clients concurrently
- it supports HTTP GET request with query and header parsing
- web client can host html file on our server

- client can submit POST request
- client supports the directory listing of a chosen directory
- client can run cgi script using the server

To execute all the above mentioned features, we created context for each required actions using `HttpServer` object and implemented `HttpHandler` abstract class to serve corresponding purpose.

How to test the system:

All the files from user side are in a specific directory of our machine. If it is used on another machine, depending on the location of the folder the user chooses to host, the path needs to be updated. User might also need to add external library to support the “com.sun.net.httpserver” API.

For our case, used a folder named `Root` which is attached with the project. The port we used is 8000.

- To transport static file, user needs to write **localhost:8000/get** (assuming he is using port 8000).
- To post, he needs to write **localhost:8000/get**. This will fetch a form, user needs to submit the form. The result will be stored in `testing.txt` file.
- For directory listing, he needs to write **localhost:8000/index**
- For launching a cgi file, he needs to write **localhost:8000/getCGI**. The attached cgi file with this system runs a dynamic date and time generator content.

This would work if user wants to run the `Root` folder that's attached with this project and changes the path in code according to his directory path. If user wants to run other files, he

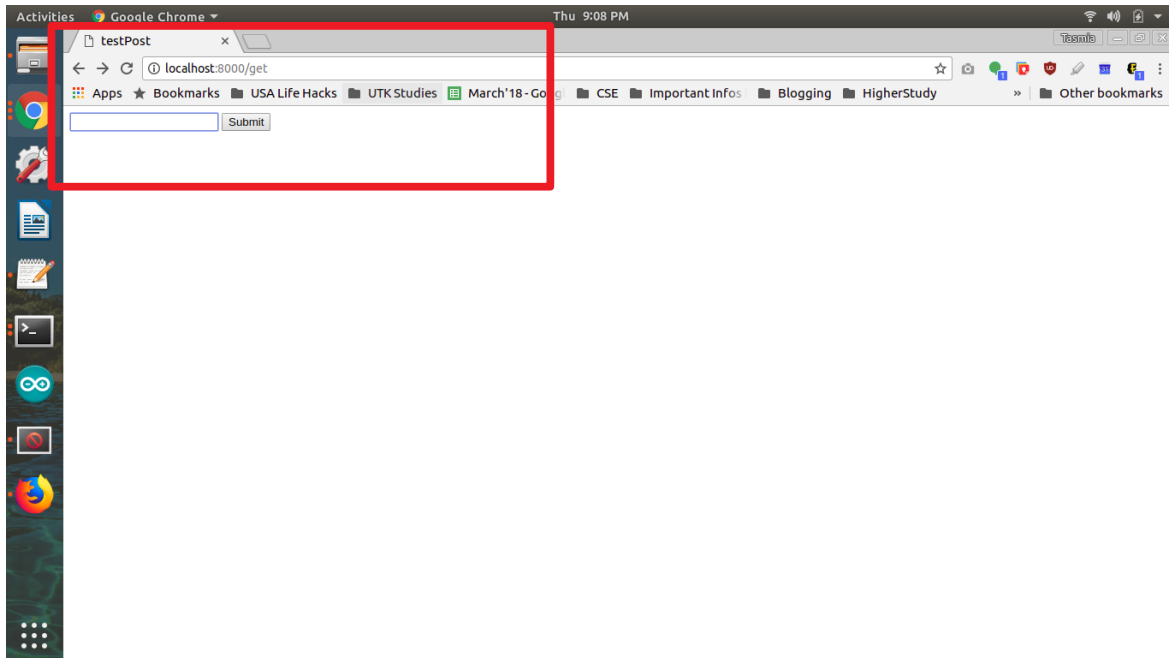
needs to change the path and file name in the corresponding class otherwise the context would call the default.

Results:

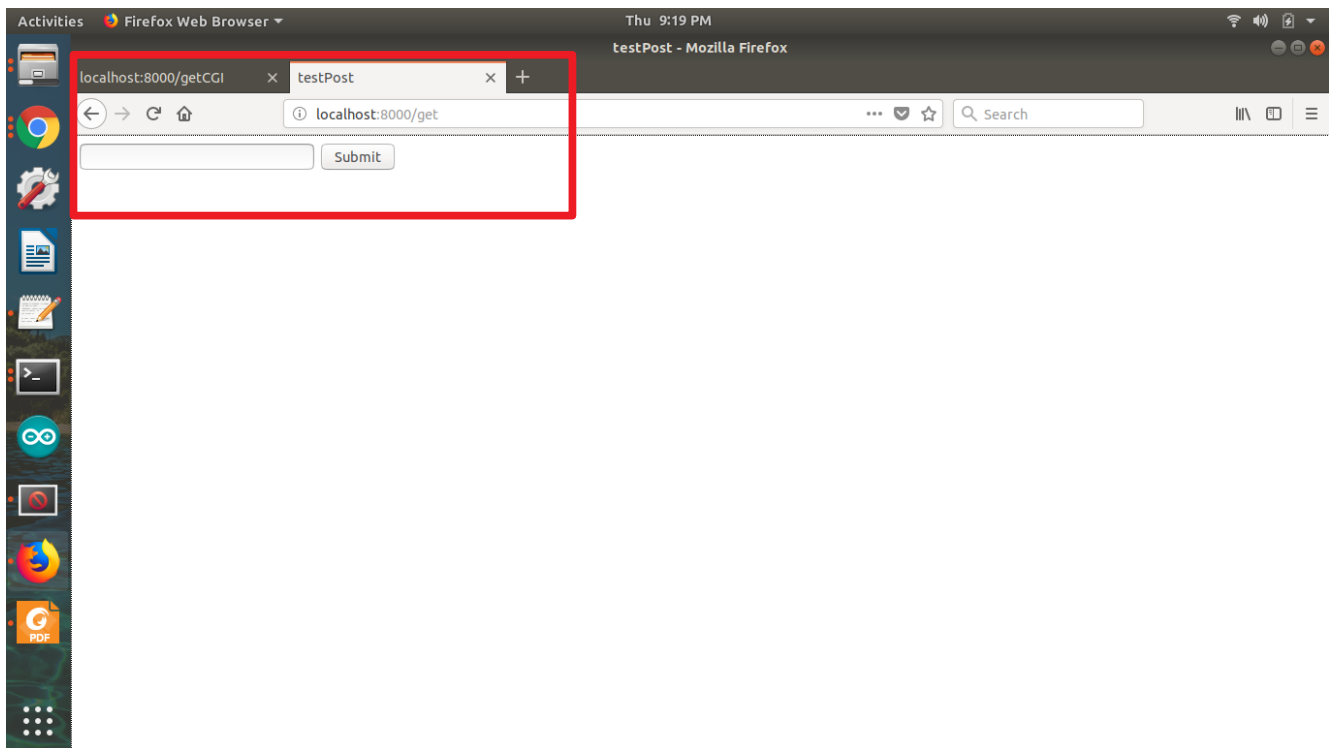
Here we will show how our server supports each of the features. We will provide corresponding screenshots below to verify the results.

GET Request supporting static file transport:

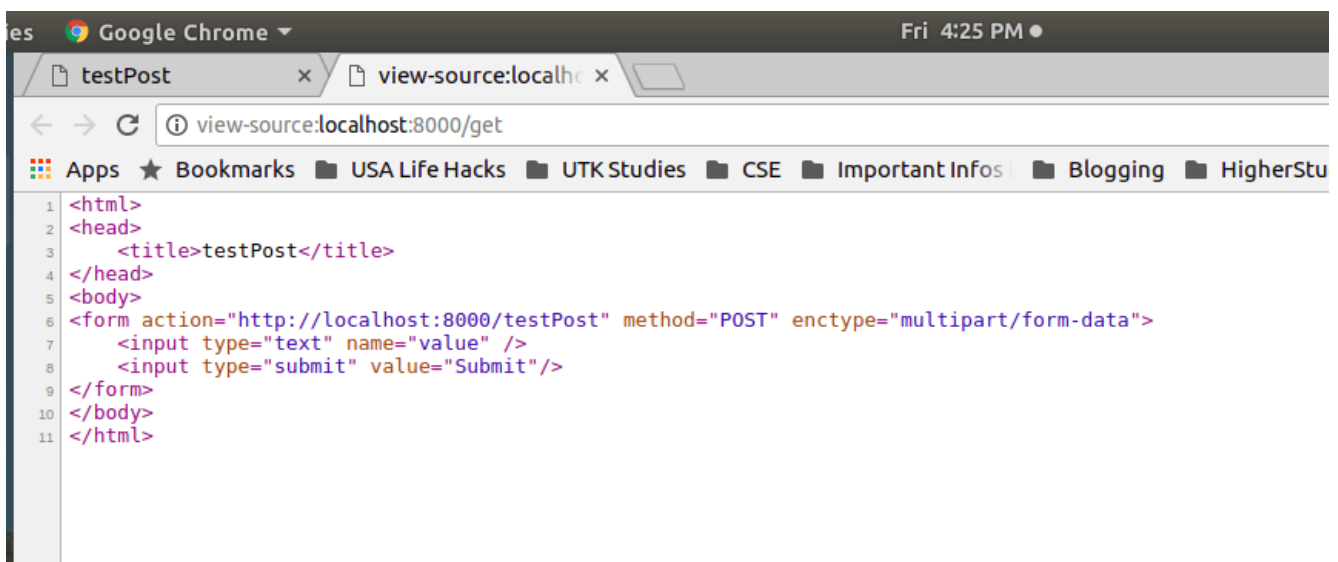
Chrome Browser:



Firefox Browser:



Inside the marked boxes for both browser we can see that our server can host an html file that's on user's machine. The html file that we ran for the experiment is as follows:

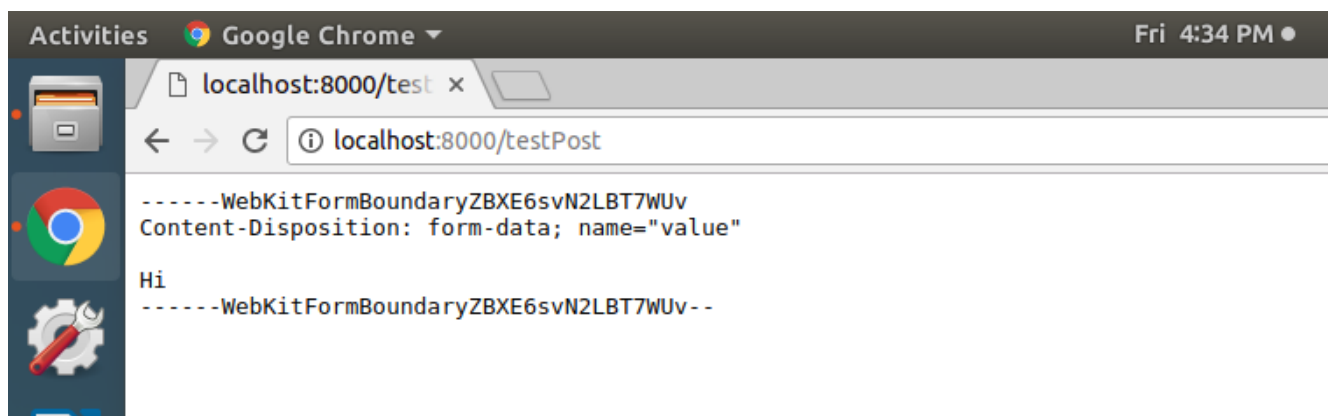
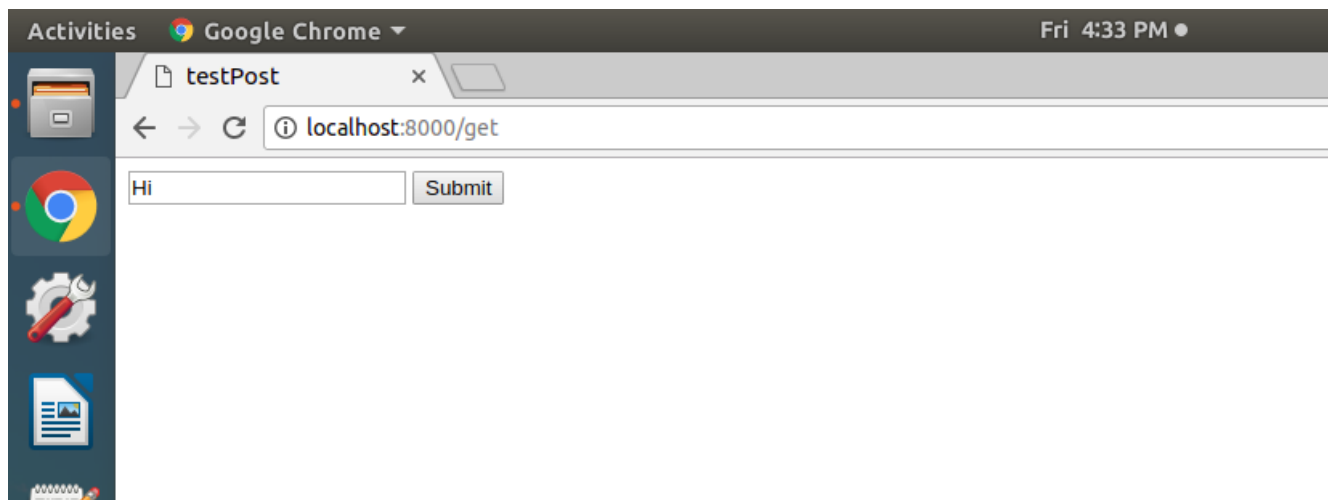


The submit button can process a POST request as you can see the method of the form is “POST”. This brings to our next feature which is described below.

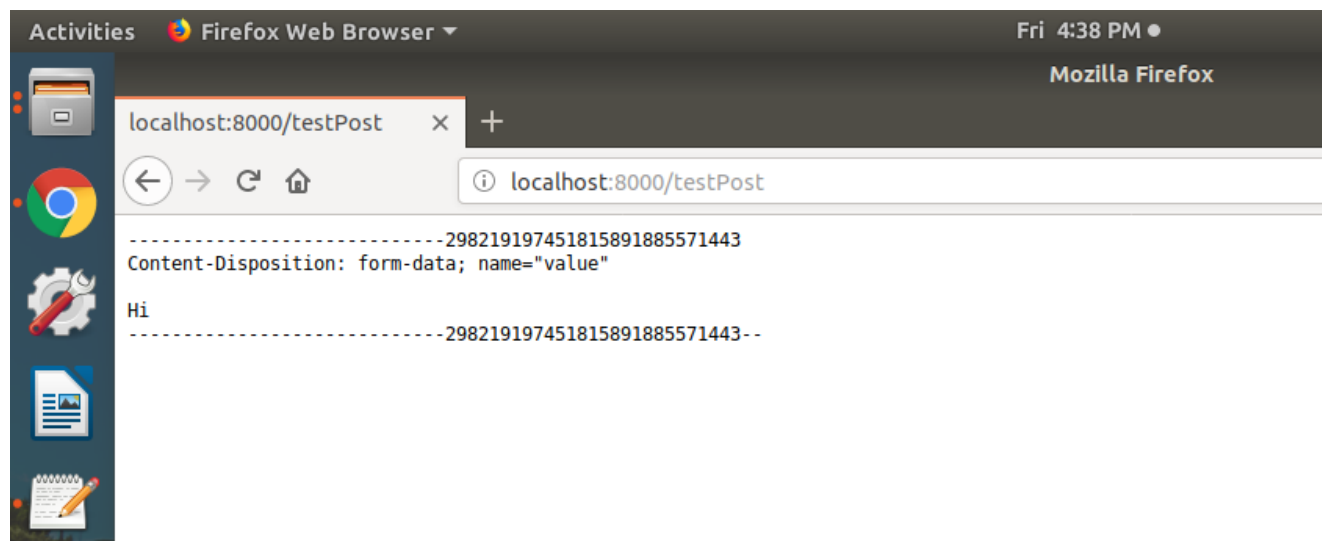
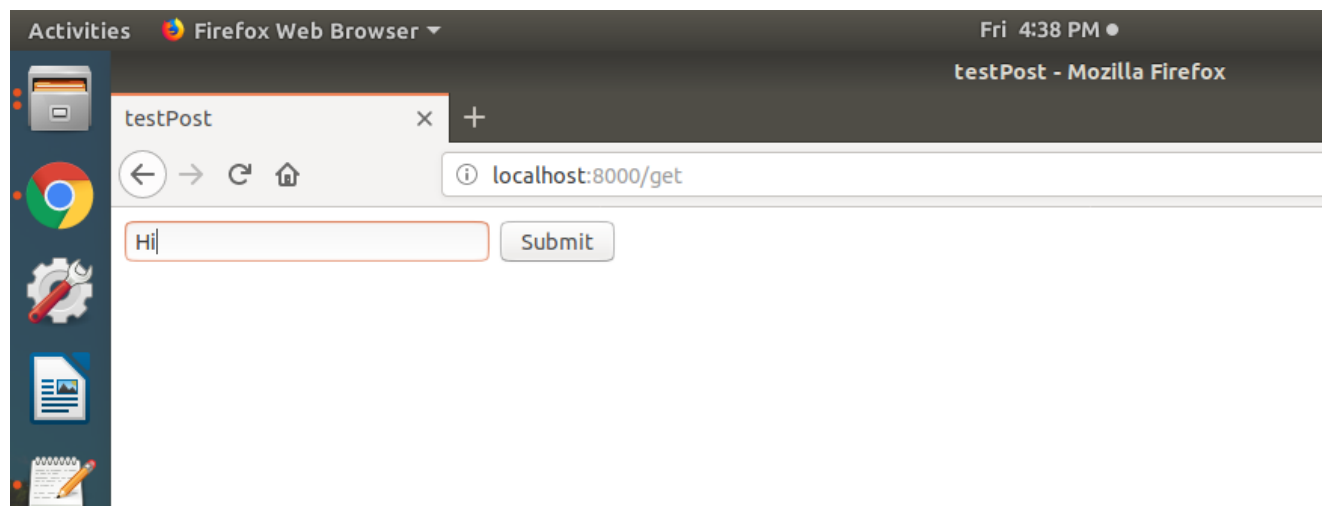
POST request:

Our server supports POST request.

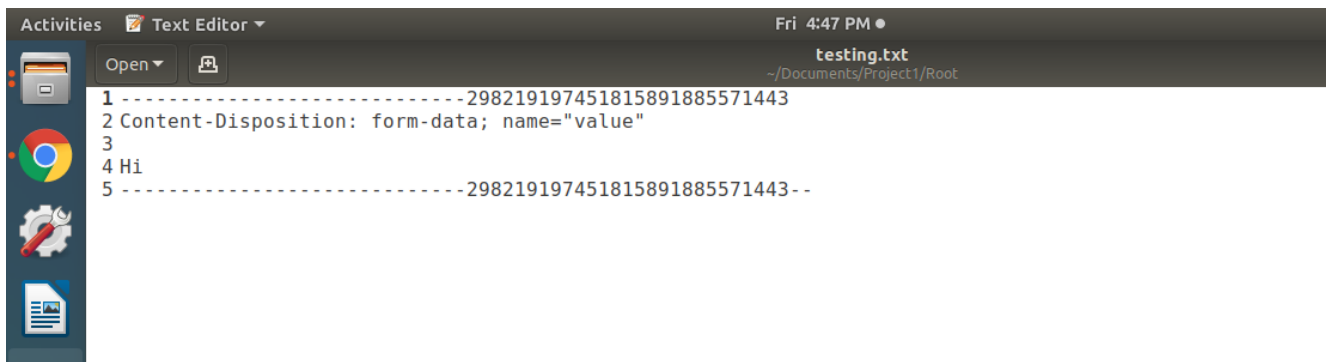
Chrome browser:



Firefox Browser:



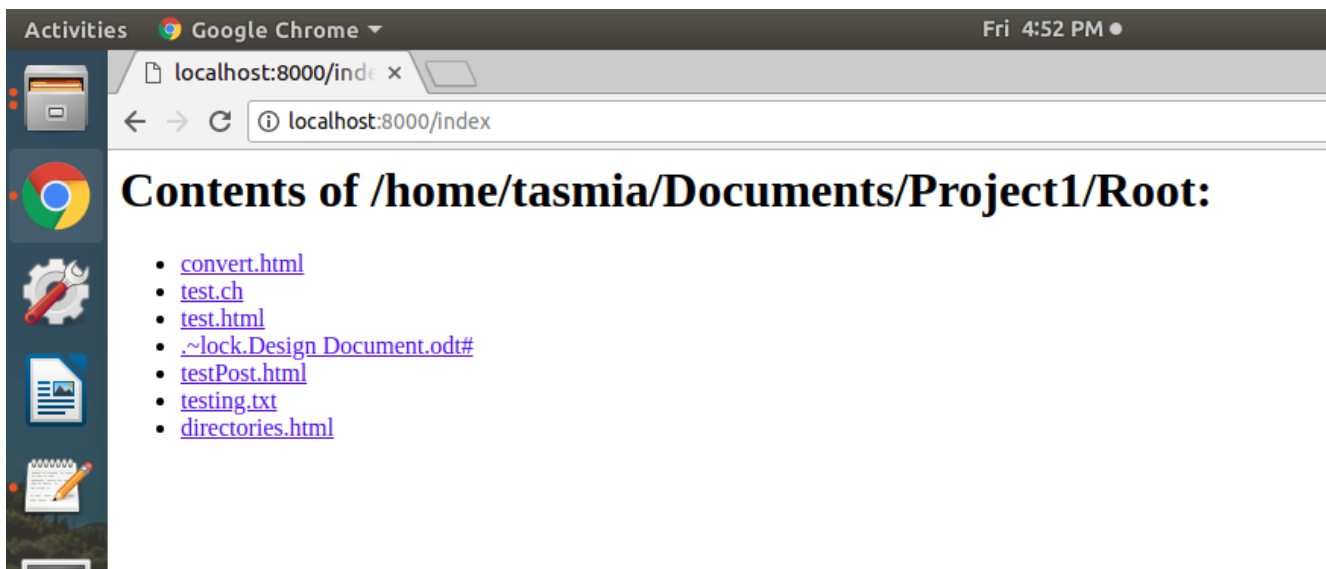
For both of the browsers when user writes a text on the text box and hits the submit button our server can process the POST request. Thus we verified that our server supports users to submit a basic HTTP form. According to the project requirement, our sever writes the result into a local file. The following screenshot can verify that.



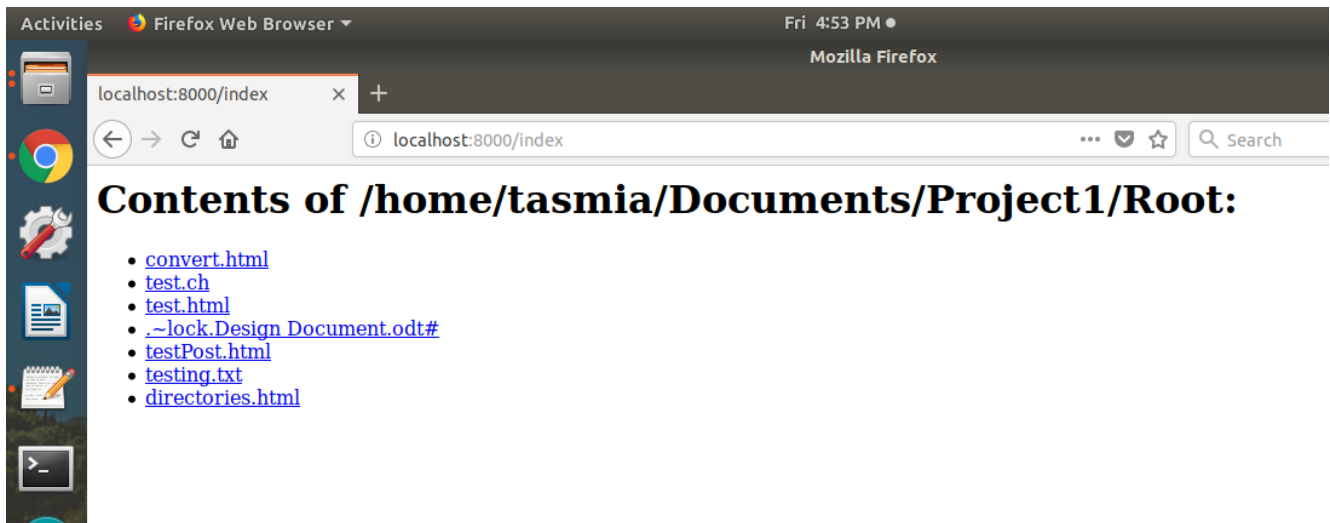
```
1 -----298219197451815891885571443
2 Content-Disposition: form-data; name="value"
3
4 Hi
5 -----298219197451815891885571443--
```

Directory Listing:

Chrome Browser:



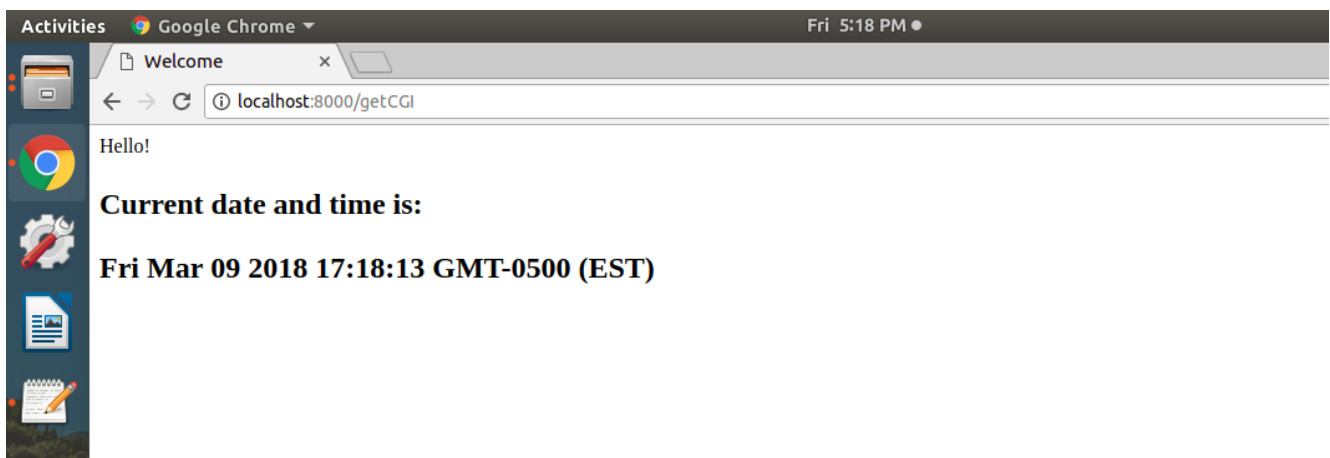
Firefox Browser:



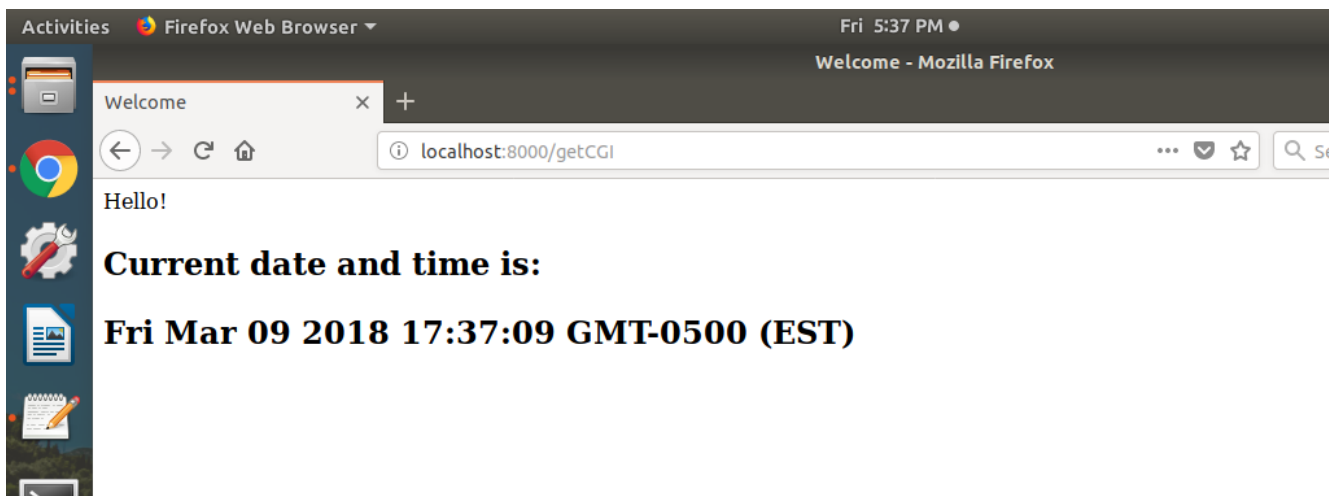
The screenshots show that user can see the directory list of a certain directory. Our server supports this feature for both the browsers. When we tested this, we specified the path of the Root directory in the “DirectoryHandler” class. That’s why the screenshot is showing the list of files within the Root folder.

CGI Script:

Chrome Browser:

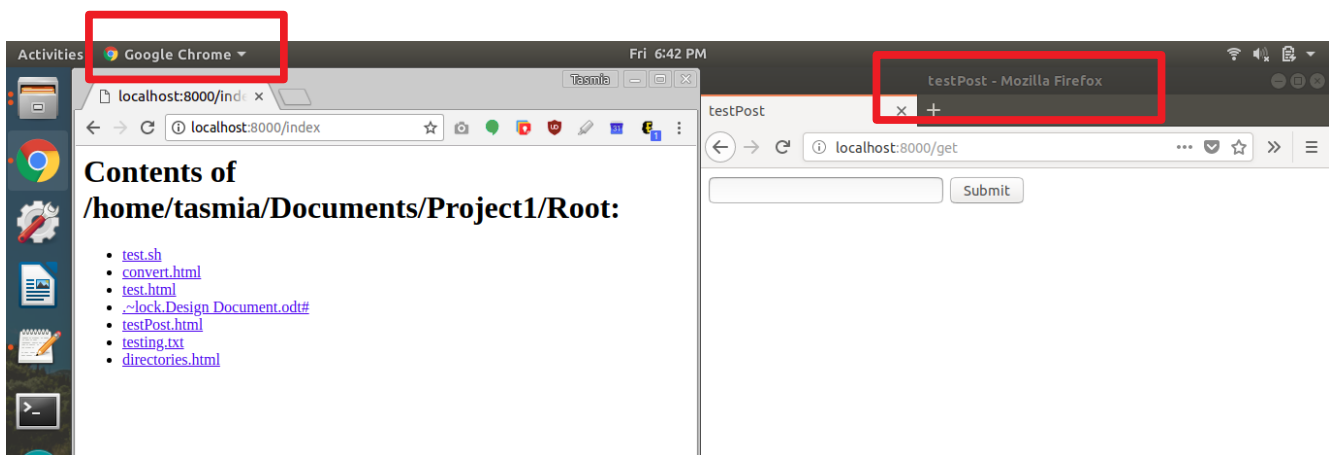


Firefox Browser:



There is a test.sh file in our Root directory which is a bash cgi script. The getCGI context calls the CGIHandler class and this class processes the test.sh file. Our server generates an html file and thus it supports basic cgi script. The test.sh cgi script generates the current date and time. On the screenshots for both the browser, we can see that /getCGI generates the processed html page and it also shows the dynamic content.

Multi Threading:



Here, the screenshot shows that our server can handle multiple web clients concurrently as both firefox and chrome browser can handle user's requests at a time while the server is running. Multiple clients can issue requests for any of the above mentioned features independently and our server can handle and support the requests concurrently.

In a nutshell, our basic HTTP server meets all the requirements.