PROGRAMMING ASSIGNMENT 1: HTTP FILESERVER

COSC560

February 28, 2019

Clara Nguyen
Rachel Offutt
University of Tennessee EECS

Contents

0.1 Project Summary

The goal is to implement a basic HTTP server that supports directory listing, static HTML files, user file uploads. The server needs to run on a Linux server environment, such as those supported in our lab machines. Specifically, your web server needs to support the following features:

- HTTP requests with query and header parsing
- HTML page navigation
- Static file transport allowing users to submit a file to the server side

0.2 Project Specific Requirements

Generally, the implementation of a HTTP server is based on socket programming in this assignment. First, the code needs to allocate a server socket, bind it to a port, and then listen for incoming connections. Next, the server accepts an incoming client connection and parse the input data stream into a HTTP request. Based on the request's parameters, it then forms a response and sends it back to the client. In a loop, the server perform steps 2 and 3 for as long as the server is running.
To observe what a web browser such as Firefox or chrome would send to a HTTP server, you can use netcat to simulate a server. For instance, run the fake server with nc -l -p 10010 and then use your web browser to go to http://hostname:10010. You should then see all HTTP request and headers the web browser sends to the HTTP server. For example, you may see something like following (using Chrome as the browser):

```
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application
      /xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US;,en;q=0.9
```

In general, a valid HTTP response looks like this:

```
HTTP/1.0 200 OK
Content-Type: text/html
<html>
...
</html>
```

The first line contains the HTTP status code for the request. The second line instructs the client (i.e. web browser) what type of data to expect (i.e. mime-type). Each of these lines should be ended with ˚. Additionally, it is important that after the Content-Type line you include a blank line consisting of only ˚. Most web clients will expect this blank line before parsing for the actual content.

Once you have implemented a webserver and verified it is working by accessing web pages using a web browser such as Firefox or Chrome, and by using command line tools such as curl or wget. For testing needs, we have provided three HTML pages as samples, namely, index.html, page1.html, and page2.html for your needs. If you place these three pages in a home directory, your server will be able to serve the index.html page by default. This page contains links that link to the second and third page, which you can use as testing benchmarks for your webserver.

This programming assignment does not specify any programming language you choose. You may choose any mainstream programming language such as Java, Python, or C to implement the requirements. You do, however, are required to provide detailed testing results in the form of screenshots on how to run your code, what results you obtain, and how you tested your results to know that they are correct.

We assume that if there is a page named "index.html" in your home folder, then it will be served as the default page. Besides static HTML pages, you need to study the HTTP protocol, such as the one available here:

https://www.w3.org/Protocols/HTTP/1.1/rfc2616bis/draft-lafon-rfc2616bis-03.html

So that you will only allow users to submit a basic HTTP form. This is usually done by using a SUBMIT button on a web page, and the web server will record the results into a local file. The details on the implementation will depend on your design choices. You need to document your design choices in detail, and provide necessary screenshots to illustrate your results.

Note that, prepackaged HTTP server classes, such as Python http.server class, should not be used for this project. Usually these classes provide much more functions than what is required here, and a project that directly calls their APIs will not be given project credits.

0.3 Project Design Choices

0.3.1 Language Choice

We chose to implement this project in Python to gain more experience in this programming language and because of how well documented Python sockets are.

0.3.2 Configuration

The application must be fine-tunable. As such, this webserver has a config JSON that is named config.json. This file is loaded at run-time and contains data on where essential directories are, as well as host and port values. This is what it may look like:

```
{
    "page_root": "./htdocs",
    "uploads": "./uploads",
    "host": "",
    "port": 28961
}
```

The following keys are required:
page_root - Root directory of all webpages.
uploads - Directory to store uploads.
host - Host address (Leave blank for "localhost")
port - Port number for server

0.3.3 Python Sockets

In this project, we are not allowed to use HTTP.sever, however, we were allowed and encouraged to use Python sockets. Since we previously decided that our programming language would be Python, the only way to complete this project would be with the use of Python sockets.

0.3.4 Threading

Modern browsers, such as Google Chrome and Firefox, send multiple requests per page load. Since the server had a single thread and was calling socket.recv to reconstruct a request at a time, it couldn't handle multiple requests being sent simultaneously.

The packets being sent were often interlaced between the multiple requests. For instance, if Google Chrome sent 2 requests of 4096 bytes each, they both are broken up into 2 groups of 4 requests of 1024 bytes each, which are sent to the server. But they are sent all in one stream, and the server gets them all in a first-come, first-serve (queue) way.
This means that the server could receive a few packets of the first request, and then a few for the second. The only guarantee is the order of the packets received for each request. So you will always get the first fragment of the first request before the second.
When a single-threaded server is trying to parse packets for the first request, it will get stuck at waiting if it gets a fragment from the second request.
The solution wasn't so obvious, and Google doesn't directly state it either. But the solution was trivial... make the server multithreaded. This allowed the requests to be split up on a per-thread basis, meaning that, while the socket.recv function was blocked on some threads, other threads were able to run it and receive bytes for their respective requests and reconstruct it.
The multithreaded solution also has the positive effect of allowing multiple users to load pages and other content simultaneously.

0.3.5 Directory Listing

If a directory does not feature an index.html file, then the server will not throw a 404, but rather create a webpage on-the-fly and return it instead. This page will list all files in the current directory, links to all of them, and also a link to go up a directory. A few issues were faced when dealing with directory listing. Firstly, if the URL is simply the directory without the final /, then all paths on the page that feature a .. will associate the directory as one higher. For instance, consider the URL: http://localhost:28961/test_dir/test. Suppose that test_dir/test is a directory.
The link in there to go to the Parent Directory is defined as ../. If it's correctly defined, it should point to http://localhost:28961/test_dir. However, because there is no final / in the URL, the link will point to http://localhost:28961/ instead.
The problem is because the browser thinks that the test is a file instead of a directory. To be fair, we do generate a page on-the-fly, so it makes sense that the browser makes this assumption. The solution is to simply slap on a / at the end of the URL.
However, we can't just go in and do a url += '/'. The browser's URL bar won't update. And even if we force it through Javascript, the links won't update. What do we do? Return Status Code 308 and tell it to redirect the page to the same page with the /

appended on.
The issue lies in the file being read in. The type of the file could be of a binary type like jpg, mp3, wav, etc. Thus, the read procedure of the server had to be rewritten to construct the HTTP response as a raw buffer of bytes. Whenever that issue was fixed, images loaded flawlessly.
As a positive side effect, MIME types such as audio/wave, audio/ogg, etc, were also implemented. These implementations may be found on the Media Test Page and Image Test Page.

0.3.6 File Uploads

The server is capable of accepting multipart/form-data entries via POST. It does this by receiving packets all at once via the following:

```
request = client_connection.recv(content_length, socket.MSG_WAITALL);
```

The procedure of reading all bytes this way allows for the server to reconstruct the file and dump it in the upload directory, specified by the configuration JSON file on the server's bootup. When a file is successfully uploaded, it will send a response to the client to load the requested page.
There is an alternate way to download files, via calling recv multiple times with a size of 1024 and reconstructing the buffer manually. However, the server has had more success with trying to receive all of the bytes at once with the socket.MSG_WAITALL flag passed in. This makes this server UNIX-only, unfortunately.

0.3.7 Matching GET and POST requests via Regex

When the server receives a GET or POST request, we want to extract the data from it to figure out what to do next. Requests like these have headers in the first line that may look like this:

```
GET /index.html HTTP...
POST /index.html HTTP...
```

As such, the following Regex expressions are utilised:

```
import re
re.findall("GET \/([^?]*?)[ ?#].*HTTP", req_text);  # GET
re.findall("POST \/([^?]*?)[ ?#].*HTTP", req_text); # POST
```

These expressions capture the path of the URL that the user is wanting to go to after the request has been processed. Usually it's for loading a page at a specified path.

0.4 Running the Project

NOTE: YOU MUST HAVE PYTHON3 ENABLED IN ORDER TO RUN THIS. PLEASE TYPE THE FOLLOWING COMMAND:

```
source /opt/rh/python33/enable
```

1. Step by step running
   - Open up a terminal in Linux with the project downloaded.
   - Change your working directory to

     `/http_server/python`
   - Type the following command:

     `sh run.sh`
   - In your web browser, type the following into your web address bar:

     `localhost:28961`
   - You are now on the web server's index.html page.
2. screenshot of run command

Figure 1: The figure above shows the run command to start the web server and the result of the webserver running.

0.5 Implementation Screenshots

NOTE: We have already demoed to the TA for this course and have proven that our implementation of the HTTP web server is correct and in working order.

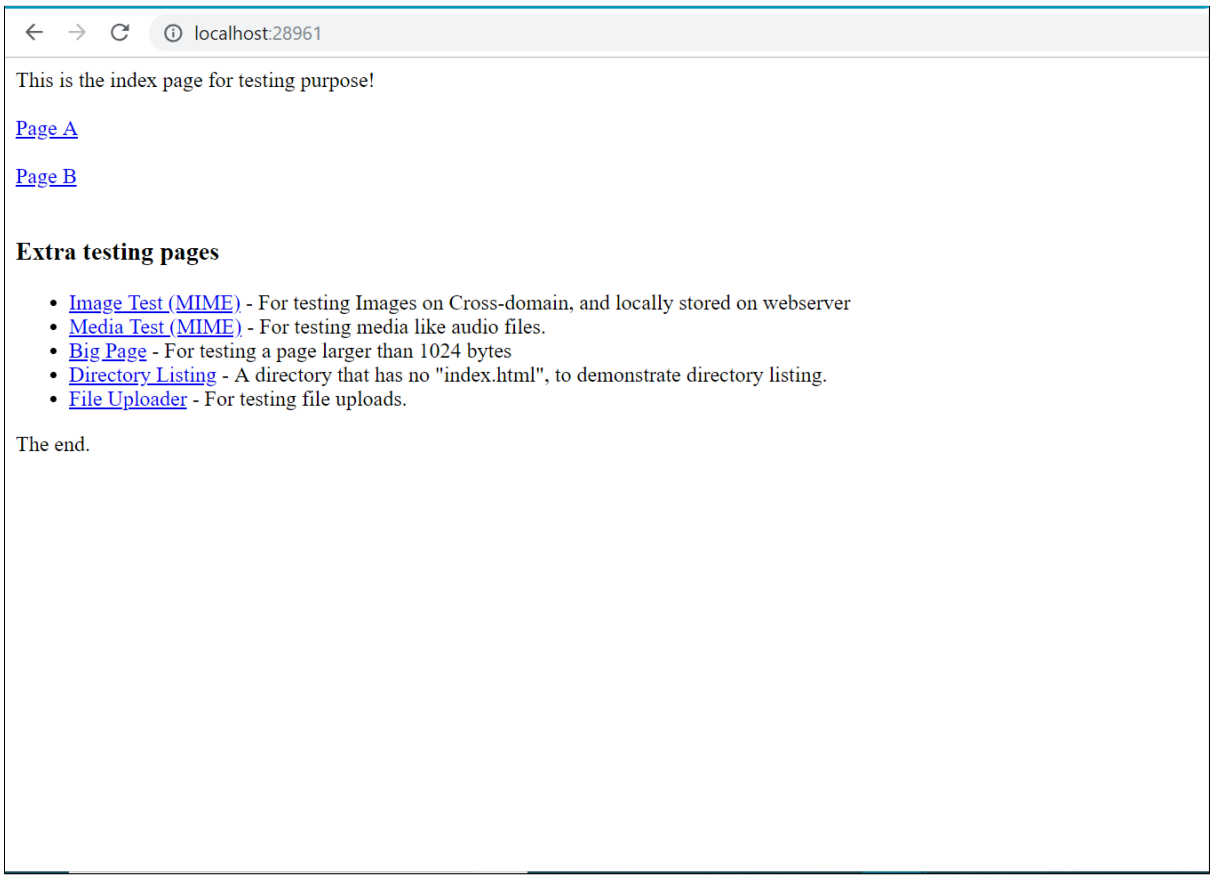

Figure 2: The figure above shows the index.html page of the webserver.

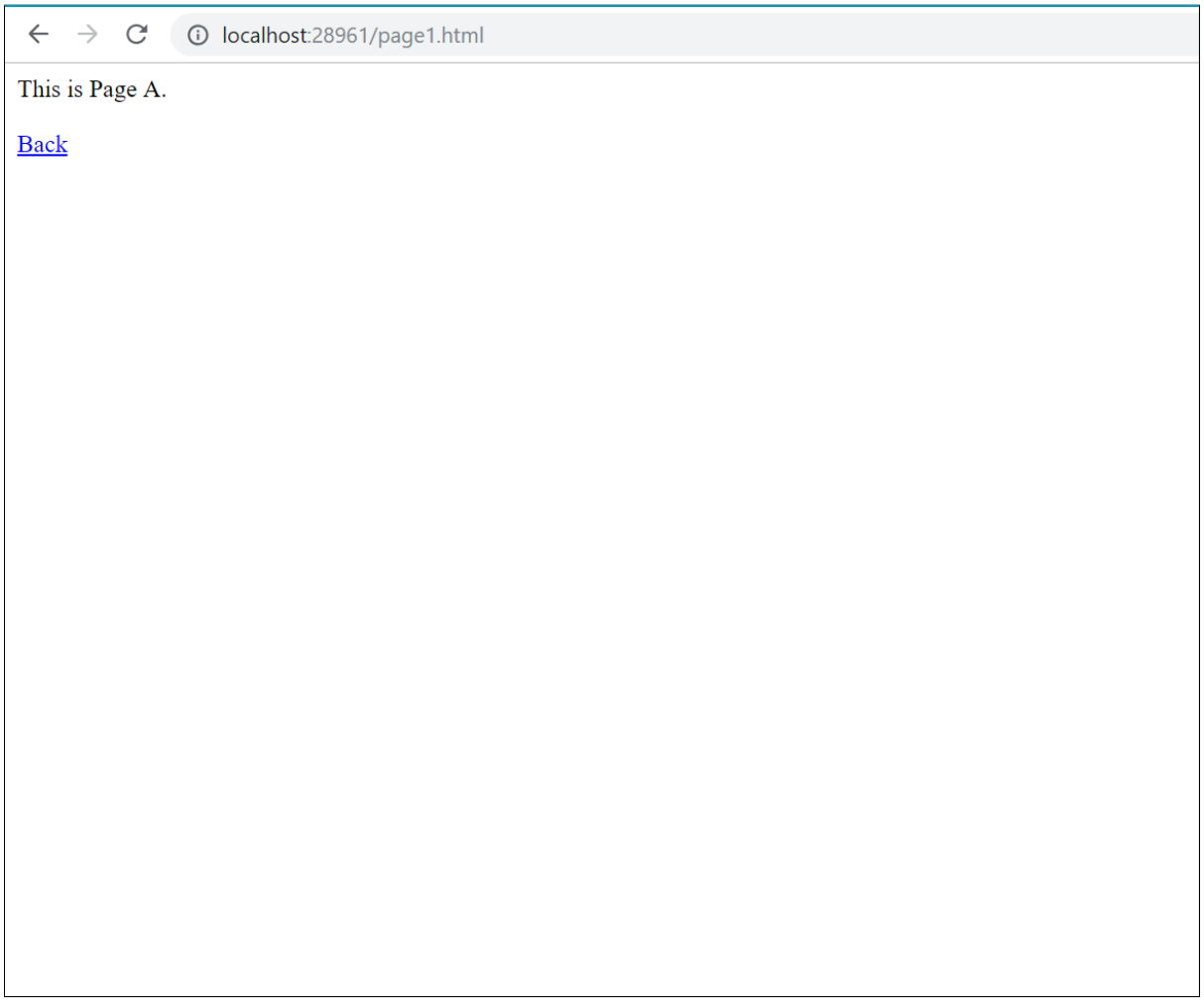

Figure 3: The figure above shows the additional page, Page A, in the webserver.

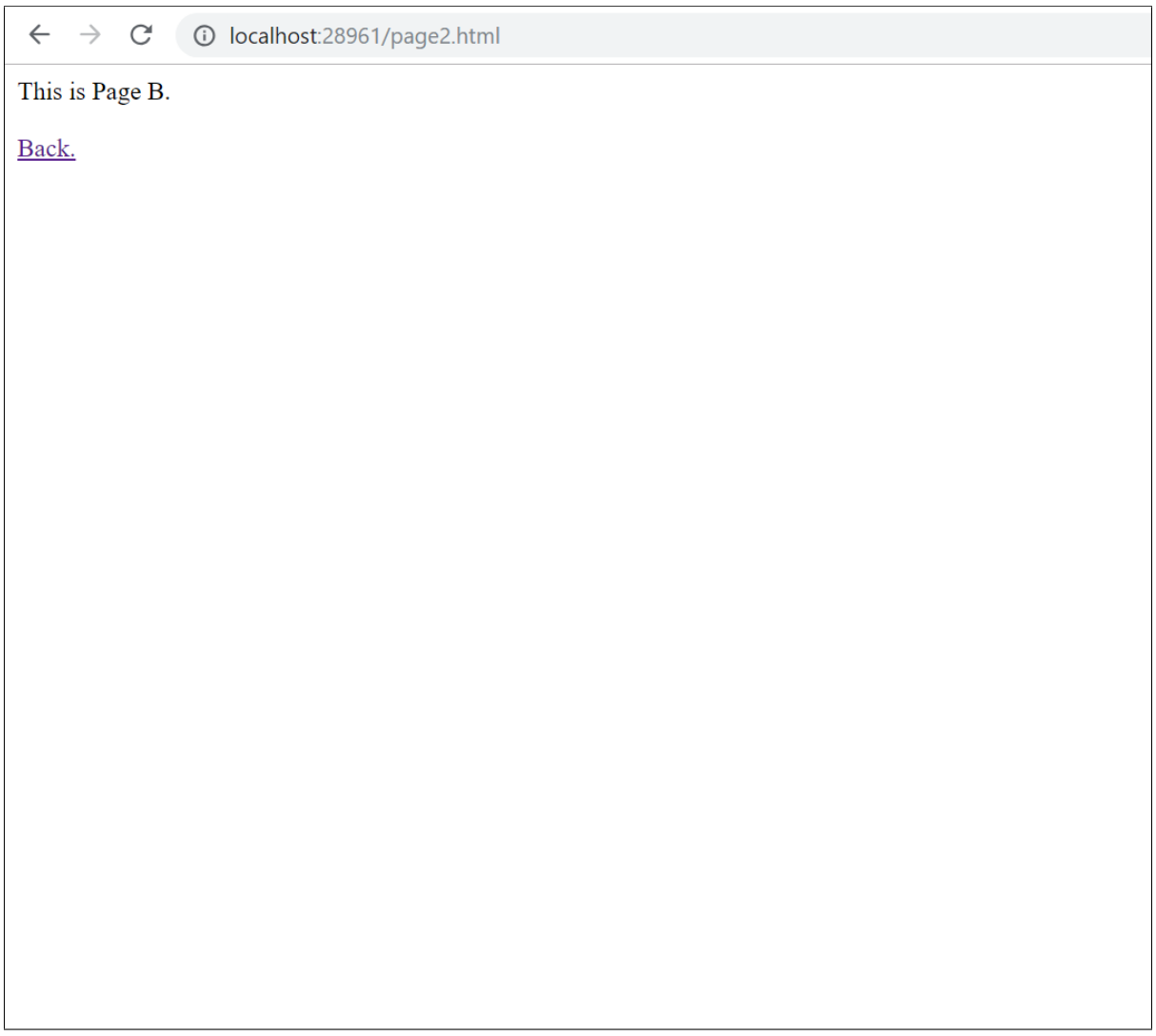

Figure 4: The figure above shows the additional page, Page B, in the webserver.

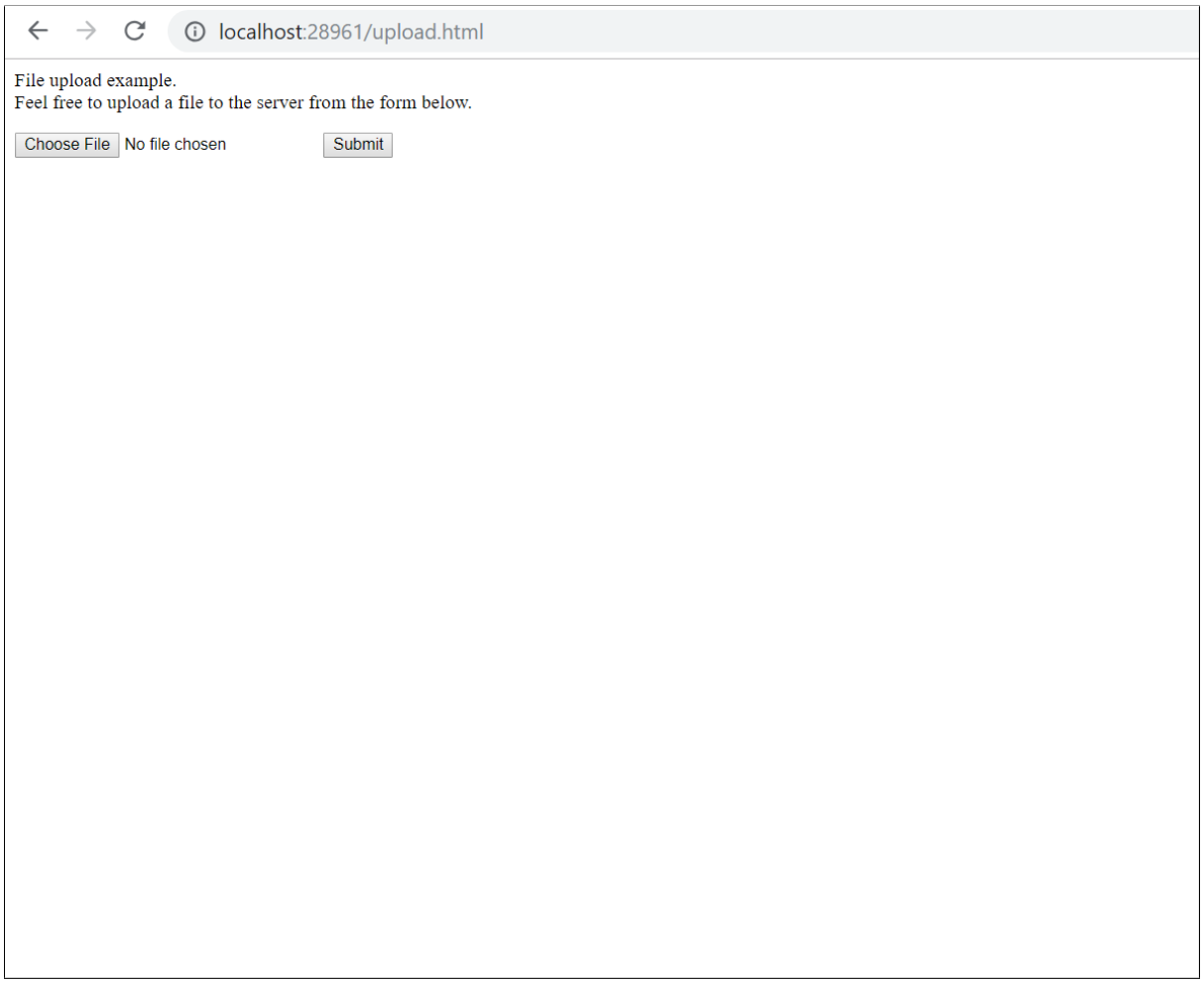

Figure 5: The figure above shows the additional page, for uploading files.

Figure 6: The figure above that the server has received the request to upload the file.

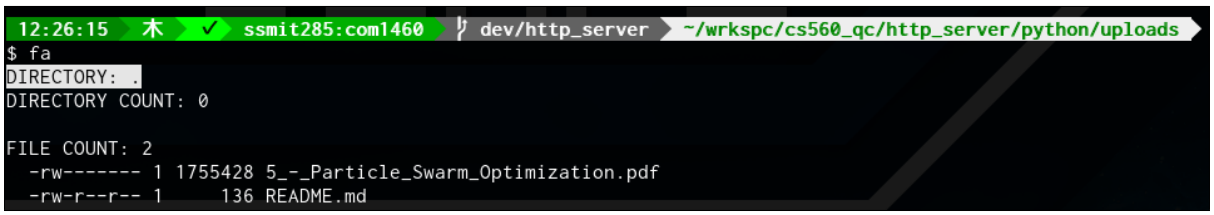

Figure 7: The figure above shows that the server has uploaded the file and the file is now listed in the directory listing.

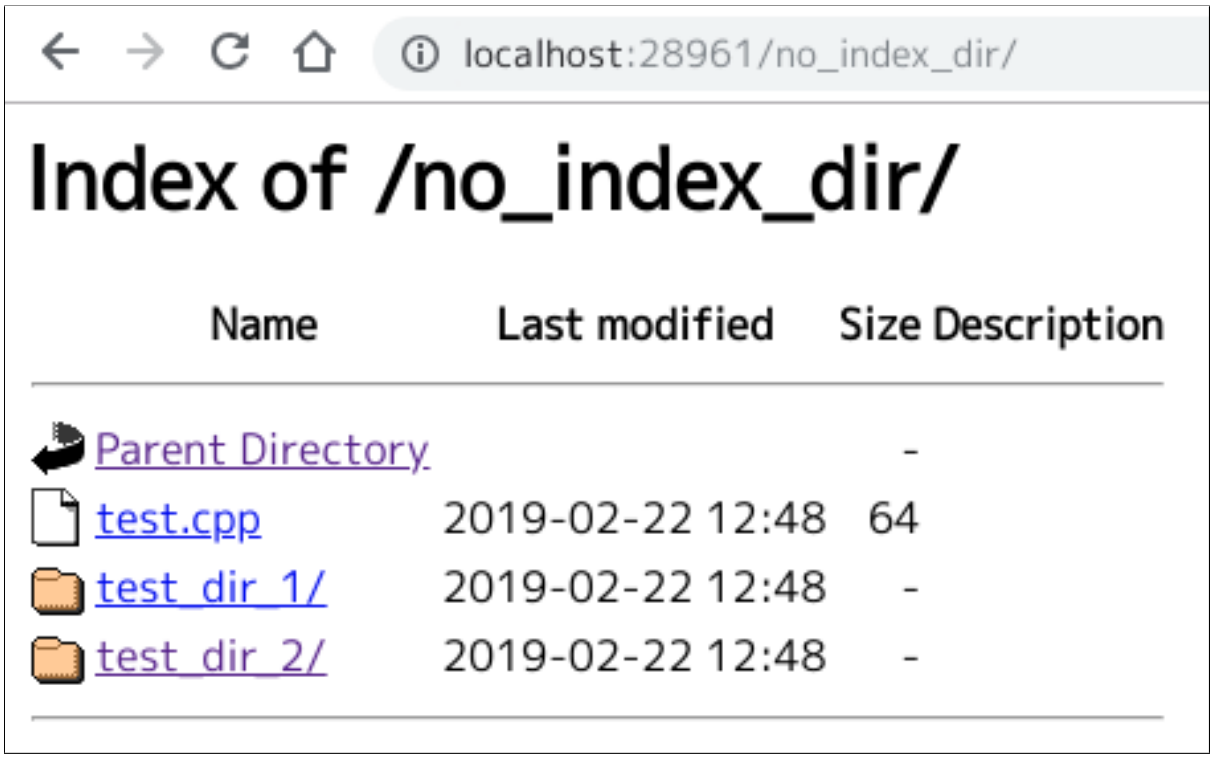

Figure 8: The figure above shows the directory listing page.

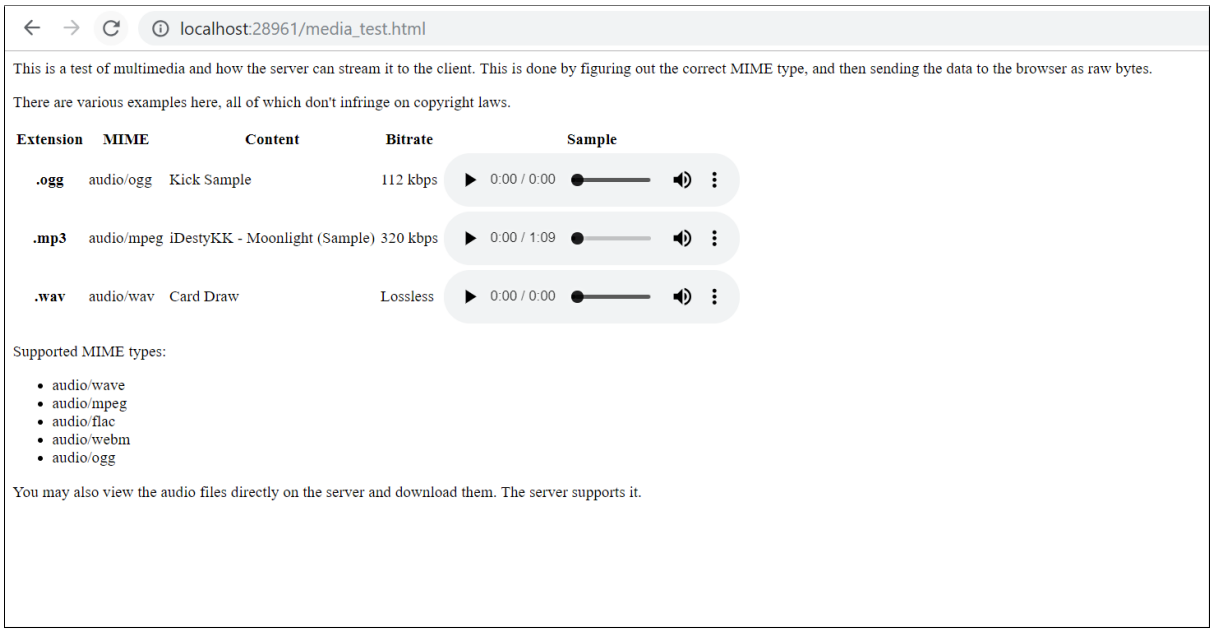

Figure 9: The figure above shows the additional page, the Media Test Page, in the webserver. This was not a main requirement of the assignment, but was added in as an enhancement.

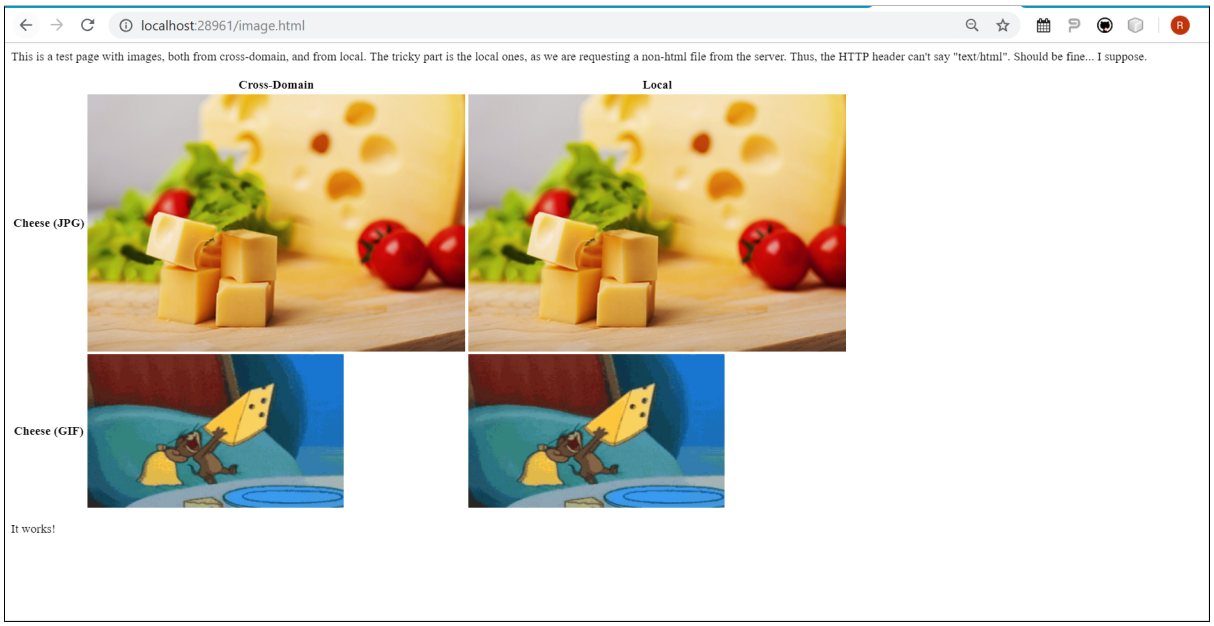

Figure 10: The figure above shows the additional page, the Image Test Page, in the webserver. This was not a main requirement of the assignment, but was added in as an enhancement. This page demonstrates locally hosted images and externally hosted images.