

Introduction:

For this project, we aimed to create a simple HTTP server that served static pages from a directory using HTTP/1.1. Our first goal was that we wanted to mimic a web server that had multiple webpages. We accomplished this by creating a folder that could hold multiple files. Each file represents a page on a web server. Another goal of ours was that we wanted our HTTP server to be able to handle requests sent from a client. To do this, we created an API that parsed the request and converted it into a struct that we defined to represent an HTTP request. This allows our server to be able to handle requests from our own client that we created as well as requests from the command line using CURL. Moreover, we specifically wanted our HTTP server to be able to handle GET and POST requests. We wanted the GET request to return the HTML for the page requested and the POST request to modify the HTML of a page. To implement the POST, we created a file that represented a sign-up page on a website. Our idea was that users would be able to send a POST request so that they could add their own name to the sign-up sheet and have it be reflected on the sign-up page when they sent a GET request. For the GET request, we wanted the HTTP server to find the file/page requested and return the contents of the file/page back to the user. We also implemented a client of our own that can send GET requests to the server. When the server receives the request, it will send back a response that will open up the HTML file on the user's webpage. This is demonstrated in our demo video.

Design/Implementation:

For this project, we built a simple HTML server and a client. To start up our server, run `./server [port] [path to folder of server]`. This will start up our server which will actively listen for connections from clients. When a client connects to our HTML server, the

server will create a connection with the client and it will create a goroutine that will handle the client's request. Currently, our server can only handle GET and POST requests. The server will check that the client's request is either a GET or a POST. It will then parse the request from the client and convert it into a struct that we defined to represent an HTTP request. If the request is a GET, the server will find the requested file and serve its content back to the client. For POST requests, our server only allows for names to be added onto the sign-up page. If the request is a POST, the server will find the sign-on page and will append the new name to the list on the sign-up page. Additionally, we created a client that can interact with our server. To start up the client, run `./client [port]`. With our client, it can send GET requests to the server. When the server responds with the HTML content, the client is able to open it up on the user's browser as demonstrated in our demo video. To test our server, we used the command line CURL as well as our own client. Commands that we used to test were:

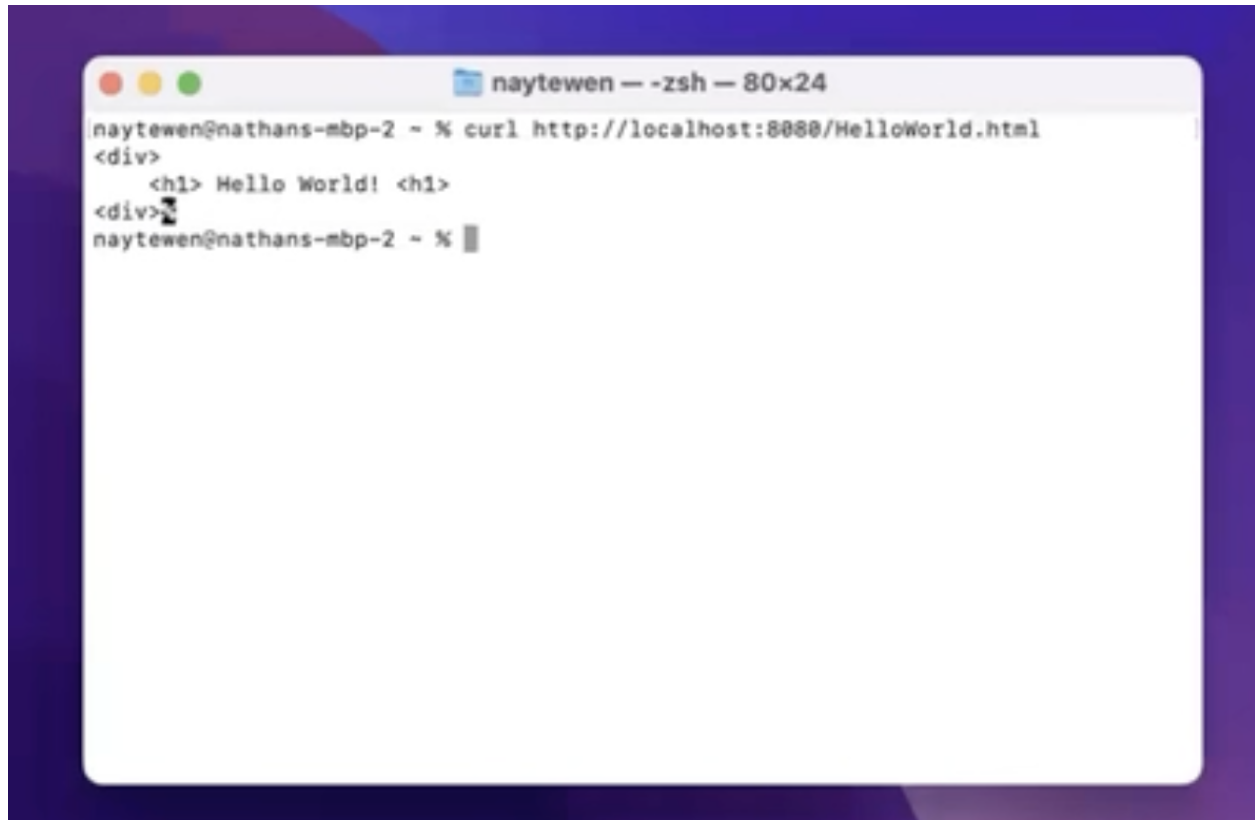
1. “curl <http://localhost:8080/SignUp.html>”
2. “curl http://localhost:8080/HelloWorld.html”
3. “curl -X POST -d "name={name}" <http://localhost:8080/SignUp.html>”

To open up the HTML content on the browser, send a request from our own client:

1. GET /HelloWorld.html HTTP/1.1 or GET /SignUp.html HTTP/1.1

Discussion/Results:

As a result of our implementation, the server can now serve static web pages when a GET request comes in, which is highlighted in the curl command below.

A screenshot of a macOS terminal window titled 'naytewen — zsh — 80x24'. The terminal shows a user named 'naytewen' at a machine named 'nathans-mbp-2' running the command 'curl http://localhost:8080/HelloWorld.html'. The output of the command is an HTML response: '<div><h1> Hello World! </h1></div>'. The prompt 'naytewen@nathans-mbp-2 ~ %' is visible at the bottom of the terminal window.

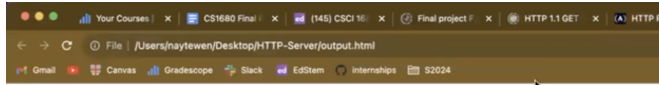
```
naytewen@nathans-mbp-2 ~ % curl http://localhost:8080/HelloWorld.html
<div>
  <h1> Hello World! </h1>
</div>
naytewen@nathans-mbp-2 ~ %
```

We also implemented a way to handle POST requests. Once a post request is received for our SignUp.html, our server will then add the name received to the list of people who signed up. The screenshots below show before and after the POST request with the key value pair “name =john” is processed.

```
naytewen -- zsh -- 80x24
naytewen@nathans-mbp-2 ~ % curl http://localhost:8080/HelloWorld.html
<div>
  <h1> Hello World! </h1>
</div>
naytewen@nathans-mbp-2 ~ % curl http://localhost:8080/SignUp.html
<div>
  <h1> Welcome to our Sign-Up form! </h1>
  <ol>
    <li> Bob </li>
    <li>nathan</li>
    <li>yuki</li>
    <li>yuki</li>
    <li>nathan</li>
    <li>nick</li>
    <li>jeffery</li>
  </ol>
</div>
naytewen@nathans-mbp-2 ~ %
```

```
naytewen -- zsh -- 80x24
  <li>yuki</li>
  <li>nathan</li>
  <li>nick</li>
  <li>jeffery</li>
</ol>
<div>
naytewen@nathans-mbp-2 ~ % curl -X POST -d "name=john" http://localhost:8080/SignUp.html
{'status' : 'success'}
naytewen@nathans-mbp-2 ~ % curl http://localhost:8080/SignUp.html
<div>
  <h1> Welcome to our Sign-Up form! </h1>
  <ol>
    <li> Bob </li>
    <li>nathan</li>
    <li>yuki</li>
    <li>yuki</li>
    <li>nathan</li>
    <li>nick</li>
    <li>jeffery</li>
    <li>john</li>
  </ol>
</div>
naytewen@nathans-mbp-2 ~ %
```

We also implemented a client program that sends a GET request to the server and automatically serves the static page using the default browser, which is illustrated below.



Welcome to our Sign-Up form!

- 1. Bob**
- 2. nathan**
- 3. yuki**
- 4. yuki**
- 5. nathan**
- 6. nick**
- 7. jeffery**
- 8. john**
- 9. freddy**

One big challenge we had was parsing the requests, as there were so many parts of an API request that's done and hidden by the curl command. We ended up printing the whole request and creating a parsing function based on that.

Conclusions/Future work:

Overall, we have learned how complex simple web browsing actually is. This project helped us learn about the logic that goes on in the backend of the server and browser. For example, in order to create the entire web browsing experience, we need to create a browser that automatically sends a GET request once the user clicks on a URL and generates dynamic pages using css and javascript once it comes back. This project was extremely fun and informative since we send requests to a server so often but we never really think about how they actually work. If we were to continue working on this project, we would make our GET and POST APIs more robust and add PUT and DELETE APIs. We would also implement more robust and functional error checking so that the server would be able to tell the client what went wrong. Additionally, we want to implement a program that puts HTML, CSS and Javascript together and serves dynamic pages to the client.