Lab 4: A Static File Server

The goal for this lab is to become familiar with HTTP and Node.js. You'll use thee JS skills we've been developing to create an HTTP server and explore the documentation for node.js.

You'll want to console.log A LOT through this lab to see what's going on and what values you're working with

Deliverable

1. 1 zip file containing:
    a. index.js
    b. package.json
    c. routes/public.js

Set Up

Before you can use Node.js on the ECC computers, you'll have to run the command

```
setup nodejs
```

This step must be done every time you start Node.js on the Linux machines. You can complete this lab on your own computer. In that case, you'll want to install node.js. The version you should install is Node.js 12. All code must be compatible with Node12, that is the version available on the ECC machines.

If you wish to install Node on your computer, please install from Node.js 12 which can be found here. It's at about Page 15 in the table. Once you've either run the setup command on the ECC machines or installed Node.js on your computer, type the following command to display the current version of Node.js. The output should be v12.22.12.

```
node --version
```

```
[aelahi@linux10625 ~]$ setup nodejs
[aelahi@linux10625 ~]$ node --version
v12.22.12
[aelahi@linux10625 ~]$
```

Writing your first HTTP server

Node.js (and most modern programming languages) include a standard library that has a lot of basic utilities. Most of those standard libraries contain an http module (or something similar).

To access the standard library in Node.js, use the built-in require function. This is similar to including another JS file in your HTML pages.

1. Create a file called index.js
2. At the top, use the require function on the http module of the standard library.

```
const http = require('http')
```

3. To create an HTTP server, use the http.createServer function.
    a. It can take one or two arguments. I'd recommend just passing in one argument: a function that gets invoked on every HTTP request.
    b. This callback receives two parameters, the first is an http.IncomingMessage object and the second is an http.ServerResponse object. I'll be using the variables req and res, respectively, to refer to these objects.
4. To start your HTTP server, use the return value from Step 3 and call its listen function with a port number.
    a. Choose a port number > 1024. The common port numbers are 3000, 8080, and 8081

```
const requestRouter = function(req, res) {}
const server = http.createServer(requestRouter)
server.listen();
```

5. If someone tries to make a request to this server, it'll just hang because we aren't sending a response. To fix that, add some functionality to the requestRouter function
    a. Assign the value 200 to the property res.statusCode

b. Call the res.end method

```
const requestRouter = function(req, res) {
  res.statusCode = 200
  res.end()
}
```

6. To run your server, open a terminal window and run the command node index.js
    a. Node.js can run any JavaScript file. But it doesn't know about all the browser related utilities like document.querySelector or fetch. Why? Because Node.js is meant for use from your terminal, not a browser.

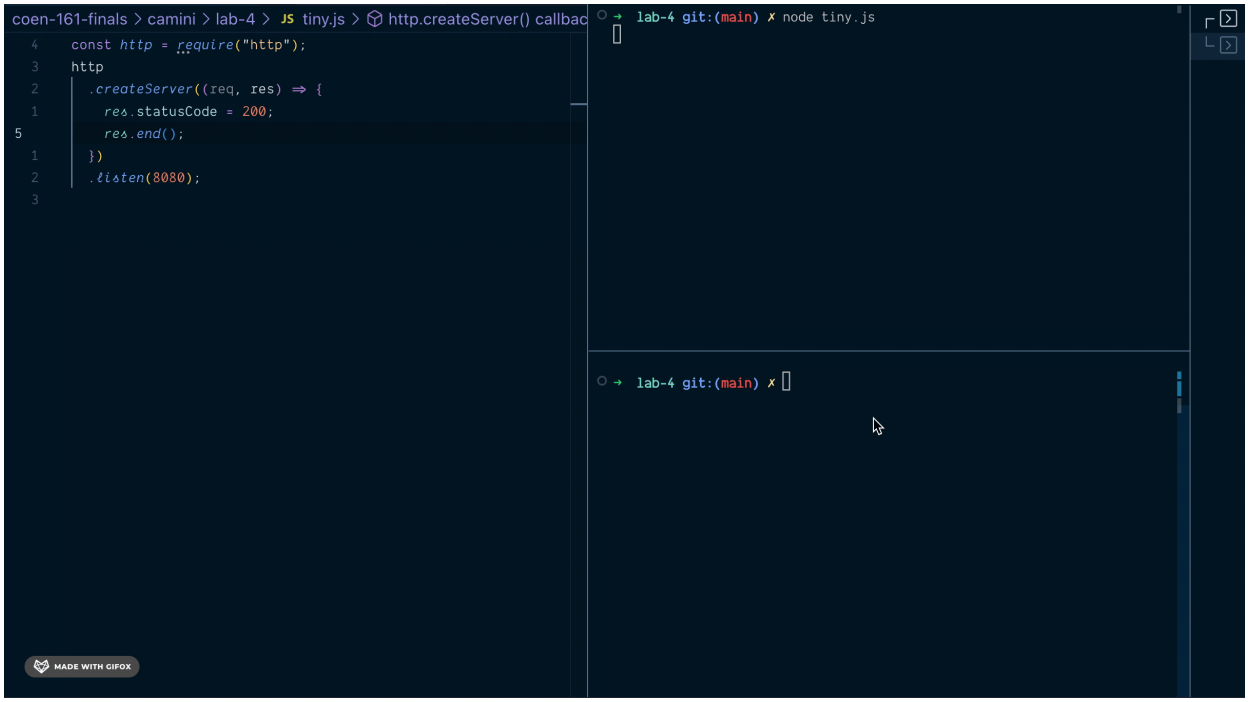
```
node index.js
```

7. Open a new terminal window on the computer you're running your server from. Type the commands below
    a. curl is a widely available UNIX utility that sends HTTP requests. Pass the address and any flags (-v stands for verbose) and you can contact your server.

```
# replace port with the number you chose

curl localhost:port
curl localhost:port -v
```

8. Demo this functionality to your TA
    a. What does the status code 200 mean?
    b. What happens if you remove the res.end call?



Writing a Generic Routing Function

This won't be useful for this week's lab. It'll be much more useful for next week's lab. But doing this now will set you up for success.

1. Create an object called routes. This will be called our RoutingTable. The keys are paths matching accepted paths in your server. The values are an object. This inner object has the keys corresponding to the accepted HTTP methods for this path and the values are the functions to be called when the IncomingMessage's path and method properties match.
    a. For now you should have an object that looks like this

```
const routes = {
  '/public/': {
    get: function(req, res) { }
  }
}
```

2. Implement the logic for routing a request based on the object created in Step 1
    a. Extract the request path (ex. '/public/') from the req object's url property.
    b. Iterate through *entries* in the RoutingTable.

        i. If the extracted path includes the current key
        ii. Check the values for the matched path.
        iii. If there is a property that matches the request.method property, call the requested function with req and res as parameters
            1. For now, the get function should always returns an OK response
        iv. If there is no matching property (ex. no routes['/public/'].post property above), then return the status code for METHOD_NOT_ALLOWED and send the response back to the user.
    c. If there is no matching path, set the response's status code to the code for NOT_FOUND and send the response back to the user.
3. Demo each of your code paths to the TA

Serving Files from Node.js

The last part of this lab is for you to start serving static files. We've so far been serving our Camini project through the school's servers. Now we'll take on that responsibility.

1. Create a folder called public in your lab-4 directory. Copy the contents of your lab 3 into the newly created public directory
2. Create a new file called routes/public.js
3. In vanilla Node.js, any file can become a module by adding a statement module.exports = <value>
    a. Create an object that mirrors the second-level object from the RoutingTable from the last step.

b. Then you can import your module by calling require('<path>')

```
// routes/public.js
module.exports = {
  get: function(req, res) {}
}

// index.js
const PublicHandler = require('./routes/public.js')
const routes = {
  '/public/': PublicHandler
}
```

4. Import the path and fs/promises libraries.
    a. path exposes functions to deal with file paths
    b. fs/promises exposes functions to interact with the file system
5. Create a variable called publicDirectory that has the value path.join(__dirname, '..', 'public')
6. Inside the get function
    a. Extract the request path from the user's request.
    b. Convert the request path to a file path based on the publicDirectory variable from above
    c. Read the request file
        i. If the file exists, set the correct status code, write the contents of the to the res object, and send the response back to the client
            1. To get images and scripts to load properly, you'll have to set the Content-Type header on the response object to the correct MIME Type for the requested file
        ii. Make sure the catch your errors
            1. When a request file doesn't exist, send back a NOT_FOUND error. You can check for err.errrno === "ENOENT" to see if the error was that the file was not found.
            2. For all other errors, set the status code to INTERNAL_SERVER_ERROR and send the response
7. Demo by making a request for your pages from the browser by typing localhost:port into the address bar of your browser. You should see your Lab 3 :D

Grading

The following table shows the point breakdown per section. Each day (including weekends) past the due date for the lab will be -5 points. You must submit your lab on Camino as well as upload it to your website.

Criteria	Points
Demo the output of Writing your first HTTP Server	20
Questions from Writing your first HTTP Server	10
Demo Generic Routing Function	30
Demo Serving Files from Node.js	40