

SIMPLE HTTP SERVER HW

This assignment is due by the date on the dropbox. Please zip up and submit your completed files to the dropbox.

Assignment:

In this assignment, you will use Node.js to build a simple HTTP server to server HTML, text and JSON. The server will accept several different URLs and make a choice of what to return based on the URL.

Getting Started:

1. Fork & Download the starter files from our Github organization.
<https://github.com/IGM-RichMedia-at-RIT/Simple-HTTP-Assignment-Start>

If you would prefer private repos, Github offers unlimited free private repos to students.

<https://education.github.com/>. If you do use private repos, **be sure** that your professor and TA/Grader have access to them!

2. Using Powershell (or Terminal on Mac/Linux), create a new node project with npm (using `npm init`). If you don't remember how to do this, refer back to *Setting Up a Node Project* on MyCourses.

For a reference for package.json options, refer to this website.

<https://docs.npmjs.com/files/package.json>

3. Install **eslint**, **eslint-config-airbnb** & **eslint-plugin-import**, using npm and **"--save-dev"**. That will include it in the package.json for me to test when grading.

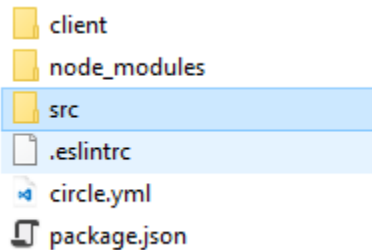
```
npm install --save-dev eslint eslint-config-airbnb eslint-plugin-import
```

Once that installs, inside your package.json, you should see a new dev-dependencies section. These are dependencies not required to run the software, but required for development.

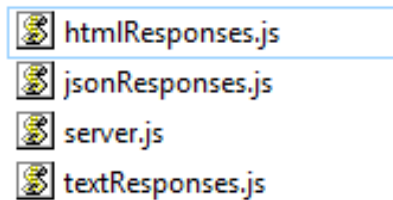
```
"devDependencies": {  
  "eslint": "^3.0.0",  
  "eslint-config-airbnb": "^9.0.1",  
  "eslint-plugin-import": "^1.10.1"  
}
```

SIMPLE HTTP SERVER HW

4. In your main folder (the only with package.json), create a folder called src. It should look like this.



5. Inside of the src folder, create files called server.js, htmlResponses.js, textResponses.js and jsonResponses.js. It should look like this.



6. Create the start, pretest and test scripts inside of your package.json file. The start script should run node on your new server.js file. That section should look like this. ESLint will check for code quality errors on any files inside of the src folder. The --fix flag will allow it to automatically fix basic errors for you.

```
"scripts": {  
  "start": "node ./src/server.js",  
  "pretest": "eslint ./src --fix",  
  "test": "echo \"Tests complete\"",  
},
```

SIMPLE HTTP SERVER HW

7. Inside of the src folder, open your server.js file and add the following.

```
const http = require('http'); |  
const htmlHandler = require('./htmlResponses.js');  
const textHandler = require('./textResponses.js');  
const jsonHandler = require('./jsonResponses.js');
```

So what does this do?

The *require* keyword is used to import modules/files. This is very similar to “using” in C#, #include in C++, or “import” in Java. These modules are either built into Node.js, are installed through npm (into the node_modules folder) or are files/modules you create.

In the first line we require the http module. This is one of Node’s built-in modules for servers. The HTTP module contains the functionality to create and operate HTTP servers. Similarly, Node has models for UDP, TCP, WebSockets and many more depending on what you want to make.

When you give *require* a module name it first checks to see if it’s built into Node. If not, it then checks the node_modules folder to see if it’s installed. Finally, it will check to see if that module is installed on the system globally (uncommon). If any of those are true, Node will pull that module into scope. Node allows you to then scope that import into a variable (which is the most common way you will see it). Since that module does not change, we make it a constant.

Occasionally, you will see modules just using *require* alone (without a variable name). This is much less common because then it imports all of those functions into the global scope of the file (less optimized and they cannot be passed around).

What happens when I require a module?

That module is instanced. The module may have support for making new objects, but the module itself is instanced. That means if I import the same module in various files, the functions of that module only get loaded into memory once. This is more performant and prevents circular dependencies. That means file A can require file B, and file B can require file A, without an infinite loop occurring.

What about the files with ./ in front of them?

SIMPLE HTTP SERVER HW

You may have noticed that the next few *require* lines we wrote have `./` and a filename. As we discussed, if you just put a module name in, Node looks it up in itself, `node_modules` or globally. However, if you put a `./` in the name and give it the path to a file, it will load that file instead.

The `./` indicates to require that it should load a file from a path instead of from the modules.

The last three lines of code we wrote are importing our files into this code and scoping them to their own variables.

8. Since we are running a server, we will also need a port to run on. The port is just a number of a specific endpoint on a computer.

[https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))

There are 65535 ports on a computer. The lower numbers are reserved for specific protocols (22 for SFTP/SSH, 80 for HTTP, 443 for HTTPS, etc).

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

We want to avoid a reserved port because other applications will probably be listening for network traffic on that port. This would cause some problems with our traffic.

Instead we will use port 3000 (well above the reserved port numbers and a common server development port).

Add this line to your code.

```
const port = process.env.PORT || process.env.NODE_PORT || 3000;
```

What does this do?

We are creating a constant for which port our server will listen on. If `process.env.PORT` is undefined/null, then we will try `process.env.NODE_PORT`. If that is undefined/null, then we set it to 3000.

What is `process.env`?

The `process` variable is a global in Node describing the running process itself. The `env` variable inside of that is the environment variables. Many servers will set their own environment variable from a config file or from another process. For that reason, we

SIMPLE HTTP SERVER HW

check process.env before setting a manual port. This is what server hosts like Heroku do in order to provide each application a different port on a given server.

9. Now let us start the server and listen for HTTP traffic. Add the following.

```
const onRequest = (request, response) => {  
  console.log(request.url);  
}  
  
http.createServer(onRequest).listen(port, () => {  
  console.log(`Listening on 127.0.0.1:${port}`);  
});
```

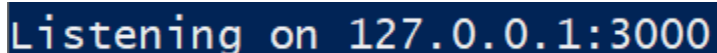
The onRequest function will be the function invoked by the HTTP server every time a new HTTP request comes in. The HTTP module will automatically pass the request and response objects to the function. The request and response objects contain all of the information about the client's request and the server's response. We can then edit the response before sending it.

The createServer method of the HTTP module will create a new HTTP server taking in the function we specified. It will then start accepting traffic on the port we specify. The callback function that we pass in as the second parameter to the listen function is called when the server successfully starts up.

10. Save this file and test your code with `npm test`. Type `npm test` into the terminal you used to install eslint earlier (double check that it's the correct folder).

You will get some *no-unused-vars* errors. Ignore those for now. We will be using those variables eventually. Fix any other errors and retest before moving on. Warnings are acceptable, but you do need to correct code errors.

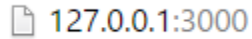
11. Start your server with `npm start`. Type `npm start` into the terminal you used to install eslint earlier (double check that it's the right folder). You should see our console.log fire.

A screenshot of a terminal window with a dark blue background and light blue text. The text reads "Listening on 127.0.0.1:3000".

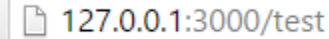
```
Listening on 127.0.0.1:3000
```

12. Now open a browser and test it out. Go to 127.0.0.1:3000, or localhost:3000. "localhost" is just an alias for "127.0.0.1". **The page will NOT load because we have told the server to respond yet.**

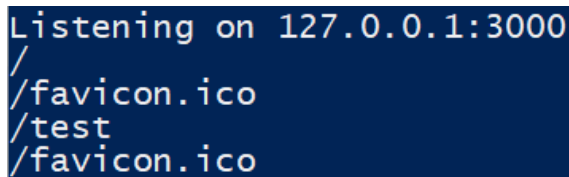
SIMPLE HTTP SERVER HW

A screenshot of a web browser's address bar. It contains the text "127.0.0.1:3000" in a blue font, with a small document icon to the left.

Try 127.0.0.1:3000/test (again the page will NOT load).

A screenshot of a web browser's address bar. It contains the text "127.0.0.1:3000/test" in a blue font, with a small document icon to the left.

Now check your Node terminal. You will probably see this. That's because any time a request comes into the server, we are printing request.url (which is the URL after domain:port on the URL bar).

A screenshot of a Node.js terminal window with a dark blue background and white text. The text shows the server listening on 127.0.0.1:3000, followed by three lines of request URLs: "/", "/favicon.ico", and "/test", each preceded by a slash. The last line is "/favicon.ico".

When you went to 127.0.0.1:3000, then the URL is automatically made /. When you went to 127.0.0.1:3000/test, the URL was /test which you will see in your terminal.

What about favicon.ico or favicon.png?

Browsers viciously want their favicon (the little icon that goes onto the page tab). This is standardized to be the URL /favicon.ico or /favicon.png (It's actually a bit more complicated than that).

With most browsers, if the browser does not receive a favicon, it will keep trying on each request (which is why you see it popped up for 127.0.0.1:3000 and for /test).

We will handle the favicons later in the semester.

13. Shut down your server for now (you can do this by hitting ctrl + C twice in the powershell/terminal window).

14. Now let's start delivering web pages. Open the htmlResponses.js file you made and add the following.

```
const fs = require('fs'); // pull in the file system module

const index = fs.readFileSync(`${__dirname}/../client/client.html`);
const page2 = fs.readFileSync(`${__dirname}/../client/client2.html`);
```

SIMPLE HTTP SERVER HW

The fs module is the file system module. You can find the API documentation here. <https://nodejs.org/api/fs.html>. Unlike JavaScript in a browser, Node.js has full access to the operating system and hardware. The file system module allows us to read and write from files.

We are using the file system module's `readFileSync` method which reads a file in synchronously.

Aren't we supposed to avoid synchronous functions?

Yes, we avoid synchronous functions because they are thread-blocking, that is they lock up the process until they complete. In a browser that means locking up the page entirely until a synchronous operation finishes. In Node, it will lock up the server process as well, which will kill server performance. Like a browser tab, Node is largely single threaded (Node has threads and clustering built-in, but that's an advanced topic).

So then why are we using a synchronous operation here?

An exception to the rule of not using synchronous operations is on startup and shutdown when we don't have any clients or interactivity. We want to ensure those files are loaded into memory BEFORE the server starts. Since our `fs.readFileSync` calls are not in a function and are just variables in that file, they will be loaded into memory as soon as Node starts up and tries to allocate of the files into memory.

That way we guarantee that before code actually executes, these files are in memory. If we did not wait, then it would be possible for the server to try to send files before they are in memory, which is bad.

Since we are reading synchronously on server startup, that does slow down the startup process. This is something to be aware of. There are more optimal patterns to do this, but it's a bit more complex than at this point.

What about `__dirname`?

The `__dirname` (no spaces between the underscores) is a global in Node of whichever folder this file is in. This allows you to specify relative paths from the current file. This is very useful when trying to load any type of file (HTML, JSON, config files, images, etc). Remember it's always the path of whichever file you are in.

15. Now that we've loaded those files in, let's write some functionality to send the files back. We'll start with the index HTML page.

SIMPLE HTTP SERVER HW

Create a `getIndex` function that accepts a request and response. These will be the request and response objects your `onRequest` function in `server.js`. We will pass them to this function.

```
const getIndex = (request, response) => {  
  response.writeHead(200, { 'Content-Type': 'text/html' });  
  response.write(index);  
  response.end();  
};
```

The response object has all of Node's HTTP Response class's methods and attributes.

https://nodejs.org/api/http.html#http_class_http_serverresponse

The `writeHead` function allows you to write a status code (200, 404, 401, etc) and a JSON object of the headers to send back. Headers are values that are standardized by a string name that the browser will interpret. Here is a list of standardized response headers.

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Response_fields

The only one we will use for this function is 'Content-Type' because the browser needs to know what type of data we are sending back. This will tell the browser how to display the data. If you send HTML as text, the browser would just display it as the raw HTML text on the screen. By setting it to 'text/html', the browser will render it as HTML instead of raw text.

Why text/html?

'text/html' is a MIMEtype / Media Type. These are also standardized so that applications can decode data differently. You can find a list of the standardized types here (categories into types starting with application, audio, image, etc).

<http://www.iana.org/assignments/media-types/media-types.xhtml>

HTML is a text type so it is listed under text. You just look up the media type for whichever media you want to send back.

Please note: Browsers do not support all of the media types listed. This list is just a comprehensive list of the standardized media types for internet software.

What about response.write?

SIMPLE HTTP SERVER HW

Node's response object allows you to write data to the response that goes back to the client. In our case we are writing the contents of the index file to it.

You may call write as many times as you need to. You may have to if you have various parts to put together or if you are reading out a buffer (such as a file stream, video stream, etc).

And response.end?

In the HTTP standard, you can only send ONE response PER request. That means you get one response. Once response.end fires, the response is sent back to the client. The response is NOT sent UNTIL you call end.

Once the end method fires and sends the response, you will no longer be able to call response.write. **You will get an exception thrown** that says that the response has already been sent **if you try to call response.write or response.end after calling end the first time**. No more data can go with it. ***That means you need to make sure all of your data is written to the response before you call end.***

16. Now let's make another function for page 2. This is also an HTML file.

```
const getPage2 = (request, response) => {  
  response.writeHead(200, { 'Content-Type': 'text/html' });  
  response.write(page2);  
  response.end();  
};
```

Exports

Since we wrote these methods in htmlResponses and server.js imports htmlResponses, we should just be able to call these functions right?

Not quite. In Node, everything is private by default. We need to mark which things in a file are public. We do this by creating a module object. Everything in this object is marked public. This is actually using the module pattern (which you may have learned about in previous classes or in other languages).

Node automatically creates this object for you by default (empty object) and puts it into this module. The object is called exports. If we want to make something public, we simply need to add it to this object.

SIMPLE HTTP SERVER HW

17. In `htmlResponses.js`, go ahead and add the `getIndex` and `getPage2` functions to the exports. We are not adding the `index` or `page2` variables because we do not need those to be public.

```
module.exports.getIndex = getIndex;  
module.exports.getPage2 = getPage2;
```

18. Back in `server.js`, let's handle the URL and decide which page to send back based on the URL the user types in. We'll just make the default be the index page, that way if the user types in something weird we will just send them the index page.

```
const onRequest = (request, response) => {  
  console.log(request.url);  
  
  switch (request.url) {  
    case '/':  
      htmlHandler.getIndex(request, response);  
      break;  
    case '/page2':  
      htmlHandler.getPage2(request, response);  
      break;  
    default:  
      htmlHandler.getIndex(request, response);  
      break;  
  }  
};
```

19. Save these files and test your code with `npm test`. **Fix any errors (other than errors related to importing your various js files) and retest before moving on. Warnings are acceptable, but you do need to correct code errors.**
20. If your server is still running from before, stop it by pressing `Ctrl+C` two times. Then restart it using the `npm start` command.

SIMPLE HTTP SERVER HW

21. Now open a browser and test it out. Go to 127.0.0.1:3000. You should see the index page.

Our first hosted web page

You will still see the url and favicon in the terminal since we are printing those.

```
Listening on 127.0.0.1: 3000  
/  
/favicon.ico
```

22. Try to go to page2. We know our code is looking for /page2 so go to 127.0.0.1:3000/page2

Our second hosted web page

```
Listening on 127.0.0.1: 3000  
/  
/favicon.ico  
/page2  
/favicon.ico
```

23. Test the default case by typing in any URL. Remember our default page should go to the index.

Our first hosted web page

```
Listening on 127.0.0.1: 3000  
/  
/favicon.ico  
/page2  
/favicon.ico  
/dankmemes  
/favicon.ico
```

24. Shutdown your Node server by hitting Ctrl+C in the terminal twice.

SIMPLE HTTP SERVER HW

25. We will now send text responses back to the client. Open up textResponses.js add the following.

This time we're adding a dynamic function that can be called to get us the current time.

```
const hello = 'Hello World';

const getTimeString = () => {
  const d = new Date();
  const dateString = `${d.getHours()}:${d.getMinutes()}:${d.getSeconds()}`;
  return dateString;
};
```

26. For the text responses, they are pretty much the same as the HTML responses except the content type is 'text/plain'. Add one that sends the time and one that sends hello.

```
const getTime = (request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' });
  response.write(getTimeString());
  response.end();
};

const getHello = (request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' });
  response.write(hello);
  response.end();
};
```

27. In this file though, we will export the response functions, but also time and hello.

This way our JSON responses will be able to use functionality from the textResponses without recoding it. We could do this in same format as we did in the htmlResponses file (where we would say `module.exports.hello = hello`, `module.exports.getTimeString = getTimeString`, etc).

Alternatively, we can simply overwrite the entire `module.exports` object with an object of our own. The below example is simply exporting the 4 things listed. You can use this syntax or the other, or mix and match.

SIMPLE HTTP SERVER HW

```
module.exports = {  
  hello,  
  getTimeString,  
  getHello,  
  getTime  
};
```

28. Now open jsonResponses.js and import textResponses.js

```
const text = require('./textResponses.js');
```

29. Create a getHelloJSON function that creates a JSON object of the hello text from textResponses.

We will also need to stringify the JSON because our HTTP response needs to be in text.

Since this is going to be a JSON response, the media type will be application/json . JSON is considered an application type (as in code), not text. Older apps might show text/json because JSON was originally a text standard.

```
const getHelloJSON = (request, response) => {  
  const helloJSON = {  
    message: text.hello,  
  };  
  const stringMessage = JSON.stringify(helloJSON);  
  
  response.writeHead(200, { 'Content-Type': 'application/json' });  
  response.write(stringMessage);  
  response.end();  
};
```

30. Create a similar function for the time.

SIMPLE HTTP SERVER HW

```
const getTimeJSON = (request, response) => {  
  const timeJSON = {  
    time: text.getTimeString(),  
  };  
  const stringMessage = JSON.stringify(timeJSON);  
  
  response.writeHead(200, { 'Content-Type': 'application/json' });  
  response.write(stringMessage);  
  response.end();  
};
```

31. Export these two functions. Feel free to try the other format, rather than sticking to the code given below.

```
module.exports.getHelloJSON = getHelloJSON;  
module.exports.getTimeJSON = getTimeJSON;
```

32. Now let's hook it all up. Back in the server, update your onRequest method to hit all of these functions.

SIMPLE HTTP SERVER HW

```
const onRequest = (request, response) => {
  console.log(request.url);

  switch (request.url) {
    case '/':
      htmlHandler.getIndex(request, response);
      break;
    case '/page2':
      htmlHandler.getPage2(request, response);
      break;
    case '/hello':
      textHandler.getHello(request, response);
      break;
    case '/time':
      textHandler.getTime(request, response);
      break;
    case '/helloJSON':
      jsonHandler.getHelloJSON(request, response);
      break;
    case '/timeJSON':
      jsonHandler.getTimeJSON(request, response);
      break;
    default:
      htmlHandler.getIndex(request, response);
      break;
  }
};
```

33. Save these files and test your code with *npm test* . **Fix any errors and retest before moving on. Warnings are acceptable, but you do need to correct code errors.**

SIMPLE HTTP SERVER HW

34. Start your server with *npm start* and test all of the pages. **Make sure they all work and correct any errors before moving on.**

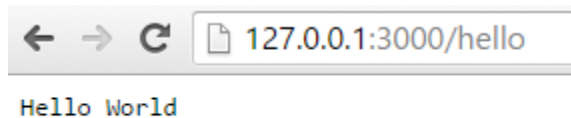
127.0.0.1:3000

Our first hosted web page

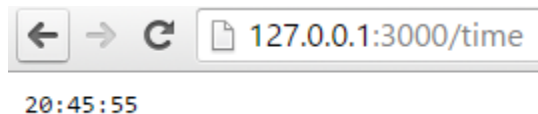
127.0.0.1:3000/page2

Our second hosted web page

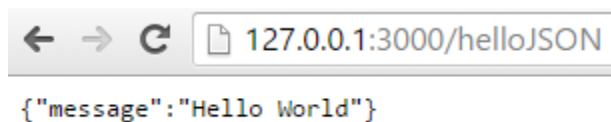
127.0.0.1:3000/hello



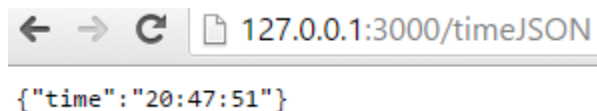
127.0.0.1:3000/time



127.0.0.1:3000/helloJSON (case sensitive)



127.0.0.1:3000/timeJSON (case sensitive)



127.0.0.1:3000/fake

Our first hosted web page

SIMPLE HTTP SERVER HW

35. Now it's your turn. I have conveniently included a spongegar.png image into the client folder.

You need to create a new file called imageResponses.js. When a user goes to the URL /dankmemes, this needs to send back the spongegar image and have it display in the browser. This will work very similarly to how the HTML functions worked since you are reading in from a file.

You'll need to look up the media type (also called MIMEtype) for PNG images for the browser to display it properly.

Test your code and the page to make sure they work before you hand it in!

36. Upload your files to your Git repo and to Heroku (see the pushing to Heroku document on MyCourses). You'll need to upload everything ***except the node_modules folder (uploading this will result in failed CircleCI tests or other issues)***.

- o Test your app on Heroku to make sure the pages work.
- o You will need to hand in your Heroku app URL in the submission.
- o **If you have problems with Heroku, please let us know. We can help.**

SIMPLE HTTP SERVER HW

ESLint & Code Quality

You are required to use ESLint with the AirBnB configuration for code quality checks. The `eslintrc` config file was given to you in the starter files. Your **server** code must pass ESLint to get credit for this part. Warnings are okay, but errors must be fixed. I won't require ESLint on the client side code in this assignment. ***If you cannot fix an error or need help, please ask us!***

You are also required to use the "pretest" script in order to run ESLint. I should be able to run "`npm test`" and have ESLint scan your code for me.

Continuous Integration

You are required to use CircleCI for continuous integration. Your build must succeed and be handed in with the assignment. The `circle.yml` config file was given to you in the starter files. CircleCI will build based on your `npm test` and ESLint configuration.

For instructions on how to do this, look at the ***CircleCI Setup*** instructions on mycourses.

SIMPLE HTTP SERVER HW

Rubric

DESCRIPTION	SCORE	VALUE %
Has the correct files – All of the files should be appropriately named and in the submission (including the client files, eslintrc and package.json). These will be needed for us to run the code.		10
Server correctly sends back index page to the browser – The page should be delivered from the server and displayed when the user goes to the URL or to any page that does not exist.		20
Server correctly sends back page2 to the browser – The page should be delivered from the server and displayed when the user goes to the page2 URL.		10
Server correctly sends back the time as text to the browser – The time should be delivered as text from the server and displayed when the user goes to the time URL.		10
Server correctly sends back Hello World as text to the browser – The text 'Hello World' should be delivered as text from the server and displayed when the user goes to the hello URL.		10
Server correctly sends back the time as JSON to the browser – The time should be delivered as JSON from the server and displayed when the user goes to the timeJSON URL.		10
Server correctly sends back Hello World as JSON to the browser – The text 'Hello World' should be delivered as JSON from the server and displayed when the user goes to the helloJSON URL.		10
Server correctly sends back spongegar image to the browser – The spongegar image should be delivered from the server and displayed when the user goes to the dankmemes URL.		20
Heroku Penalty – If your app is not functional on Heroku, there will be a penalty.		-5% (this time)
CircleCI Penalty – If your build is not successful on CircleCI, there will be a penalty.		-5% (this time)
ESLint Penalties – I should be able to run 'npm test' and have ESLint scan your code without errors. Warnings are acceptable. Check your code with ESLint often during development! If you cannot fix an error, please ask us!	1 Error	-5%
	2 Errors	-10%
	3 Errors	-15%
	4 Errors	-20%
	5+ Errors	-30%
Additional Penalties – These are point deductions for run time errors, poorly written code or improper code. There are no set values for penalties. The more penalties you make, the more points you will lose.		
TOTAL		100%

Submission:

By the due date please submit a zip of your project to the dropbox along with a link to your Heroku App (not the dashboard but the working web link from 'open app'), a link to your Github repo **AND** a link to your CircleCI build in the submission comments. In the zip include your package.json, .eslint file, src and client. You do not need to include the node_modules folder as that is generated by the package.json.