STREAMING MEDIA HW

This assignment is due by date on the dropbox.

Assignment:

In this assignment, you will use Node.js to stream basic video and audio content.

Getting Started:

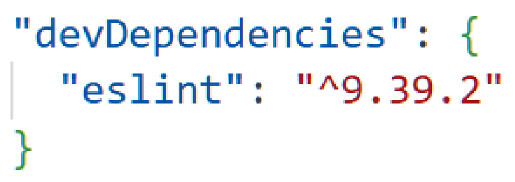
1. Fork and clone the starter files for this assignment from our Github organization.
   https://github.com/IGM-RichMedia-at-RIT/streaming-media-assignment

2. In the same folder as the starter files, create a new node project with NPM. If you don't remember how to do this, refer back to *Setting Up a Node Project* on MyCourses.
   For a reference for the package.json options, refer to this website. Not all fields are necessary.
   https://docs.npmjs.com/files/package.json

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

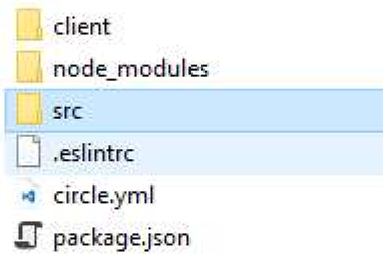
   npm install --save-dev eslint

   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. Note: dependency version may not match the one in the image below.

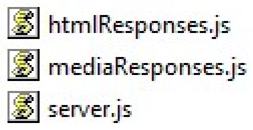
   

```
"devDependencies": {
  "eslint": "^9.39.2"
}
```

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 and mediaResponses.js. It should look like this.



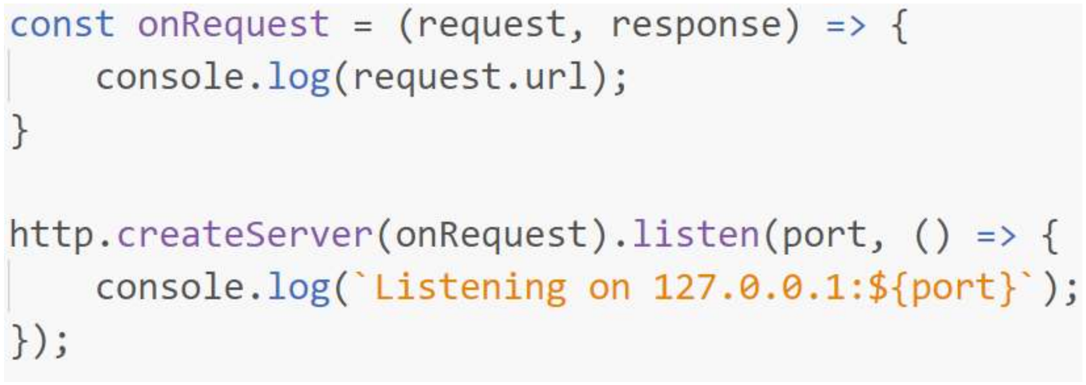
6. Create the start, pretest and test scripts inside of your package.json file. The start script should run node on your new server.js file. That section should look like this. ESLint will check for code quality errors on any files inside of the src folder. The --fix flag will allow it to automatically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

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

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 mediaHandler = require('./mediaResponses.js');

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

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

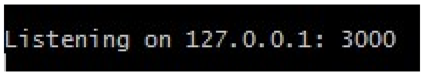


```
const onRequest = (request, response) => {
    console.log(request.url);
}

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

9. 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).

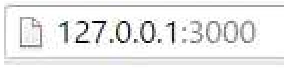
   You will get some no-unused-vars errors. Ignore those for now. We will be using those variables eventually. Fix any other errors and retest before moving on. ***Warnings are acceptable, but you do need to correct code errors.***

10. 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.



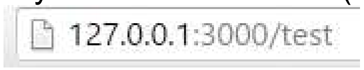
```
Listening on 127.0.0.1: 3000
```

11. Now open a browser and test it out. Go to 127.0.0.1:3000. **The page will NOT load because we have told the server to respond yet.**



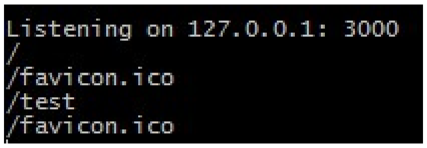
127.0.0.1:3000

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



127.0.0.1:3000/test

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).



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

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

13. Now let us start streaming some media. Previously we looked at hosting static files, but hosting large files such as videos or audio can be too memory intensive for both the server and browser. Instead, we need to *stream* larger files in order to keep memory low and processing efficient. Streaming just means that we are sending the data in pieces, rather than all at once.

    In the htmlResponses.js file, add the following. *Remember __dirname* has two underscores.

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

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

    Are we not supposed to use synchronous functions?

    For now, we are using synchronous functions for small static files, but for our larger files we will not be able to. Later we will start handling small static files in an asynchronous way as well.

14. 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();
};

module.exports.getIndex = getIndex;
```

    As a reminder: The writeHead function allows you to write a status code (200, 404, 401, etc) and a JSON object of the headers to send back.

15. 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.

We'll be adding more to this in a bit.

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

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

    switch(request.url) {
        case '/':
            htmlHandler.getIndex(request, response);
            break;
        default:
            htmlHandler.getIndex(request, response);
            break;
    }
}

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

16. Open client1.html and client2.html in the client folder of the starter code. Notice the video and audio tags. The video's party.mp4 link will call the URL */party.mp4* and the audio's bling.mp3 will call the URL */bling.mp3*. These calls are made automatically when the browser tries to load the video/audio. We will need to hook up those URLs.

client1.html

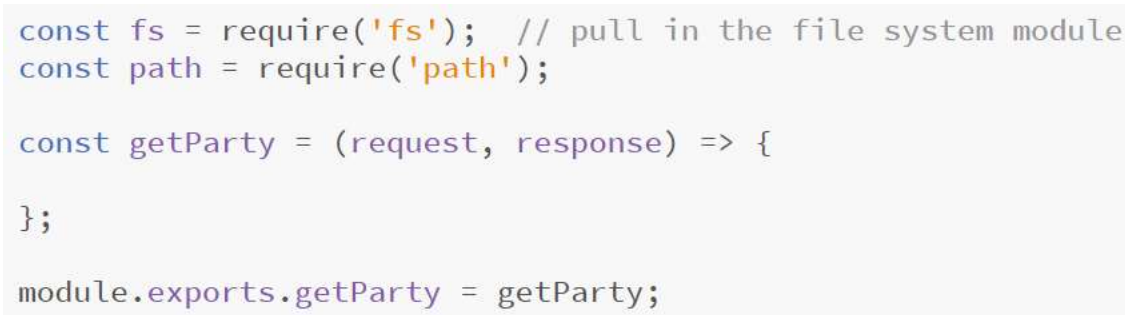
```
<body>
    <video src="party.mp4" autoplay controls></video>
</body>
```

client2.html

```
<audio controls autoplay>
  <source src="bling.mp3" type="audio/mpeg">
  Your browser does not support the audio tag.
</audio>
```

17. Now open mediaResponses.js and add the following.

Node's <u>path</u> module is a collection of utilities for working with files and paths. This will allow us to create a File object from a file path.



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

const getParty = (request, response) => {

};

module.exports.getParty = getParty;
```

18. Inside server.js, add a case for the URL */party.mp4* in the switch statement. Have it call the getParty function we just added.

19. Now inside of getParty, use the path module's resolve function to create a File object. The resolve function takes a directory and the relative path to a file from that directory. This does NOT load the file, but just create a File object based on that file.

    The fs module's <u>stat</u> function provides statistics about the file. This is an **asynchronous** function. The stat function takes a file object and a callback function of what to do next.

    This is one of our first times working with asynchronous functions in Node. Most of Node is intentionally asynchronous for performance reasons. When the stat function loads the file, it will then call the callback function that has been passed in.

    The callback of this function receives an err field and a stats object. If the err field is not null, then there was an error. In that event we will respond with an error. If the error code is 'ENOENT' (**E**rror **No** **Ent**ry), then the file could not be found. We will set the status code to 404. In the event of any error, we will send the error back to the client for now.

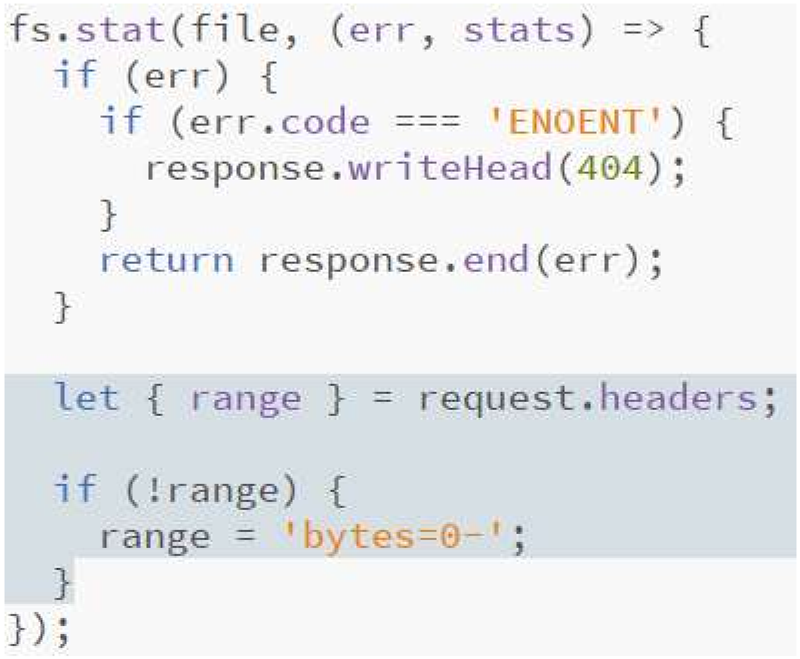
```
const getParty = (request, response) => {
  const file = path.resolve(__dirname, '../client/party.mp4');

  fs.stat(file, (err, stats) => {
    if (err) {
      if (err.code === 'ENOENT') {
        response.writeHead(404);
      }
      return response.end(err);
    }
  });
};
```

Later in the course, we will look at better error handling and avoiding these embedded callbacks.

20. After the error check, we need to see if the client sent us a range header. If there is not a range header for this request, then we will assume starting at the beginning of the file (or byte 0). Alternatively, you could return the response after setting a 416 error (Requested Range Not Satisfiable) error, but that's not as user friendly.

    Requests to stream media are sent with a range header that requests a byte range of the file (the bytes representing the part of the media they want). As the file streams or the user moves around in the file time, new requests will be sent to the server asking for a new range to add to the browser's buffer.

    To keep memory low and processing efficient, we will only send the bytes requested and only if they are valid. This means not loading the entire file into memory, but only loading a particular range of bytes from the file.



```
fs.stat(file, (err, stats) => {
  if (err) {
    if (err.code === 'ENOENT') {
      response.writeHead(404);
    }
    return response.end(err);
  }

  let { range } = request.headers;

  if (!range) {
    range = 'bytes=0-';
  }
});
```

The code “let { range } = request.headers; is making use of a feature of ES6 javascript known as a destructuring assignment. Essentially what this code says is “grab the range element out of the request.headers object, and store it in a new variable I am making called range”.

21. Now we need to grab the byte range from the request’s range header. The header range will come as bytes=0000-0001 where 0000 is the beginning and 0001 the end range of the bytes in the file. Those number will vary depending on the byte range requested.

    Often when doing video streaming like this, the client will not know the end range yet, so it will look like bytes=0000- and there will not be an end range.

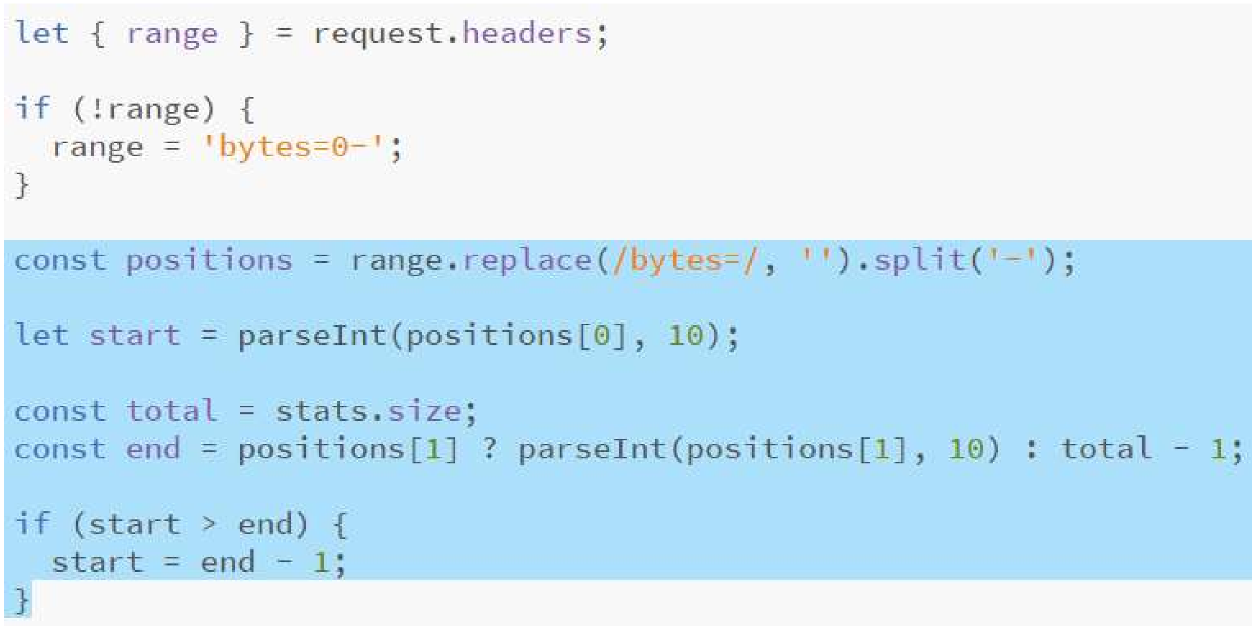
    Example of byte range header without the ending range (common)
    bytes=63995904-

    What we need to do is grab the string, replace the word bytes= with nothing and then that will give us 0000-0001. Then we can split on the **-** to get an array of beginning and end positions. ['0000', '0001']. Alternatively, you could find another way to split/parse the bytes= string from the range header.

    Next we'll parse the first position (starting range) to an int. The second parameter of parseInt is which number base to use. 10 means base 10 which is typical human readable numbers and what we'll need for the file.

    Stats.size will give us the total file size in bytes.

Next, we just need to check if we got an end position from the client. If not, we will just set our end position to the end of the file. If so, we will parse it into base 10. In the event that the start range is greater than the end range, we will need to reset the start range.



```
let { range } = request.headers;

if (!range) {
  range = 'bytes=0-';
}

const positions = range.replace(/bytes=/, '').split('-');

let start = parseInt(positions[0], 10);

const total = stats.size;
const end = positions[1] ? parseInt(positions[1], 10) : total - 1;

if (start > end) {
  start = end - 1;
}
```

22. Now we will need to determine how big of a chunk we are sending back to the browser (in bytes).

    For a streaming file we need to send back the <u>206 success code</u> (partial content). This tells the browser that it can request other ranges (before or after), but it has not received the entire file.

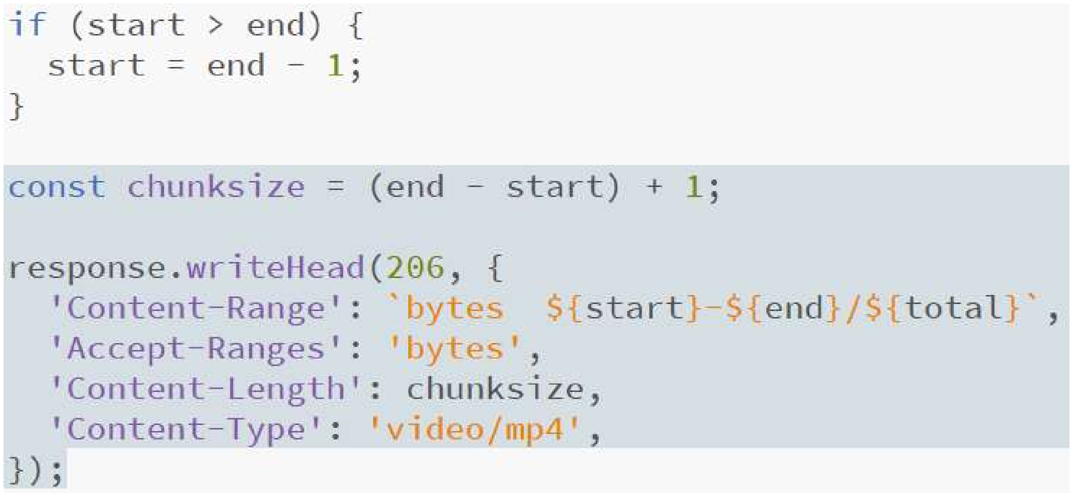
    In order for this to work, we will need to set a few headers for the client.

    Content-Range means how much we are sending out of the total. Oddly enough content-range is sent as the string "bytes 0000-0001/0002".

    Accept-Ranges tells the browser what type of data to expect the range in. Oddly enough, the only current options are 'bytes' and 'none'.

    Content-Length tells the browser how big this chunk is in bytes.

    Content-Type tells the browser the encoding type so it can reassemble the byte correctly.



```
if (start > end) {
  start = end - 1;
}

const chunksize = (end - start) + 1;

response.writeHead(206, {
  'Content-Range': `bytes  ${start}-${end}/${total}`,
  'Accept-Ranges': 'bytes',
  'Content-Length': chunksize,
  'Content-Type': 'video/mp4',
});
```

23. Next we will need to create a file stream. File streams take a File object and an object containing the start and end points in bytes. This way we only load the amount of the file necessary.

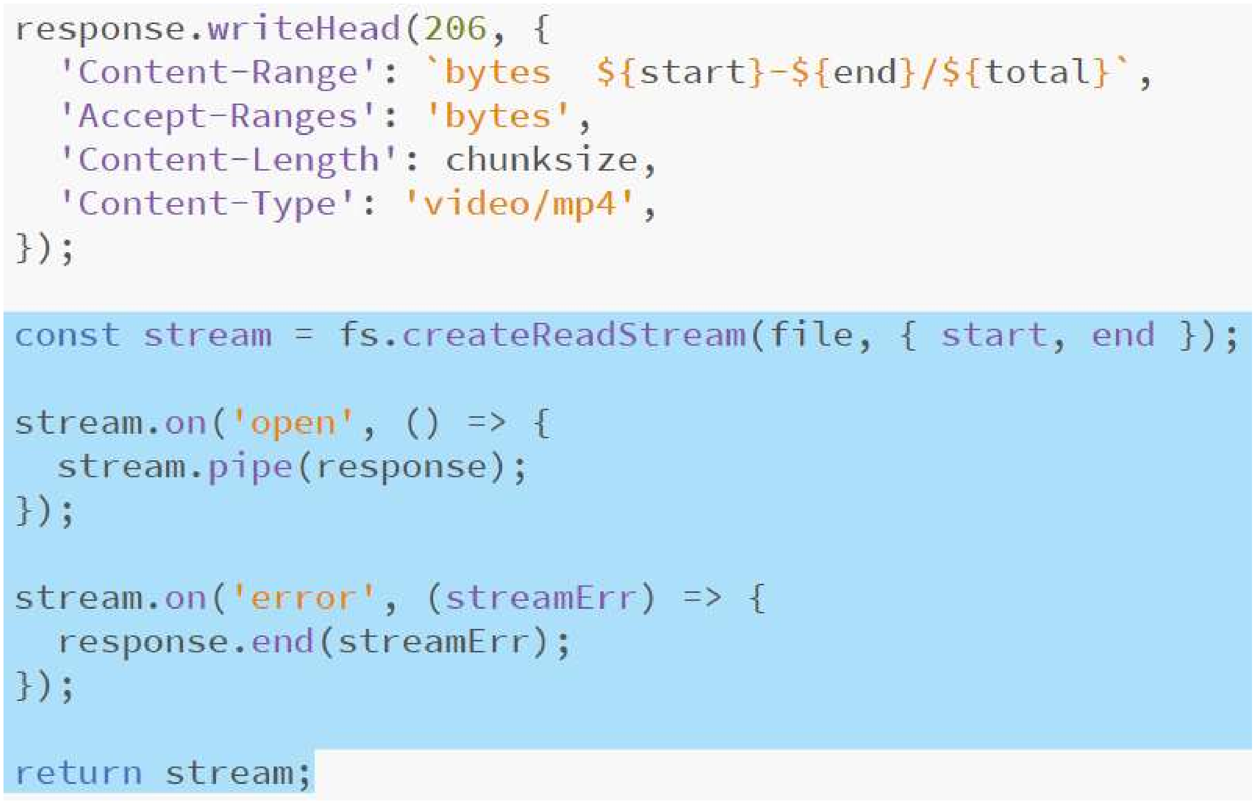
    { start, end } creates an object with the variable names set to the same as the ones passed in. For example { start, end } will yield the object { start: start, end: end }.

    Since streams are asynchronous, we'll need to provide callback functions for when the stream is in the open or error states.

    When the file opens, we will connect the file stream to our response with the stream's pipe function. The pipe function is a stream function in node that will set the output of a stream to another stream. In our case, ***this is key*** to keep it lightweight.

    We are piping the file stream directly to our client response. This means as one byte is read in from the response, it is written back to the client. Since it only reads a few bytes, sends them, and replaces them with new bytes from the file, our memory usage stays very low.

    In the event of an error (usually running out of bytes), we will end the response and return our stream error. This will tell the browser to stop listening for bytes.



```
response.writeHead(206, {
  'Content-Range': `bytes  ${start}-${end}/${total}`,
  'Accept-Ranges': 'bytes',
  'Content-Length': chunksize,
  'Content-Type': 'video/mp4',
});

const stream = fs.createReadStream(file, { start, end });

stream.on('open', () => {
  stream.pipe(response);
});

stream.on('error', (streamErr) => {
  response.end(streamErr);
});

return stream;
```

24. Save your files and test your app with *npm test*. If you have any errors, correct them before moving on.

25. Start your server with *npm start*. Go to 127.0.0.1:3000 in the browser and test the page. If it is successful, the client1.html page should load and play the video.
If it does not, then check the node window and the browser console for errors.
You will want to check your server.js to make sure you are calling the correct function. You'll also want to check your functions to make sure everything is being called.
If the video does play, then you should be able to skip around in the video safely.
In fact, you can open up multiple tabs and play the video at different points simultaneously.

26. Now it is your turn. You will need to hook up the other video and audio file, but first you should look at the code in mediaResponses.js. The code is poorly written (intentionally). You should refactor the streaming code into a reusable function to make it easy to stream other files arbitrarily.

STREAMING MEDIA HW

You need to:

- Refactor the streaming code into a cleaner reusable function that can stream an arbitrary file.
- Add functionality that sends back the client2.html page when users go to /page2
    - Note: this html page contains an audio tag that will try to load /bling.mp3 from the server.
- Create client3.html in the client folder.
    - This page should have a video tag that loads /bird.mp4 from the server.
- Add functionality that sends back the client3.html page when users go to /page3

Note: A common misconception is that by sending the html pages to the client, they'll automatically have access to the audio and video files. That is incorrect.

Look back at what we did for the client.html page and the party.mp4. In server.js, we have a handler for the '/' url (which loads the html page). When the browser loads the client.html page, it sees the video tag in that page. The video tag has a src of "party.mp4". The client only knows about files the server gives it. Servers don't work by default like your computers file system or banjo.rit.edu.

Since the browser doesn't know where party.mp4 is, it makes a GET request to the server for /party.mp4, which hits our onRequest function in server.js and gets routed to getParty in the mediaHandler.

When you go to hook up /page2 and /page3, you'll also need to hook up the resources they are trying to load (bling.mp3 and bird.mp4). In the end, your onRequest switch should have 7 cases (including the default).

Ideally, you can get your mediaResponse methods down to at least something as reusable as the below image. **Your files do NOT need to look like this.** This is just an example of what your calls *could* look like after refactoring the streaming code.

```
const getParty = (request, response) => {
  loadFile(request, response, '../client/party.mp4', 'video/mp4');
};

const getBling = (request, response) => {
  loadFile(request, response, '../client/bling.mp3', 'audio/mpeg');
};

const getBird = (request, response) => {
  loadFile(request, response, '../client/bird.mp4', 'video/mp4');
};
```

27. Upload your files to Heroku (see the pushing to Heroku document on mycourses). You'll need to upload everything except the node_modules folder. Test your app on Heroku to make sure the pages work.

    You will need to hand in your Heroku app URL in the submission.

    If you have problems with Heroku, please let us know. We can help.

ESLint & Code Quality

You are required to use ESLint with the js/recommended configuration for code quality checks. The eslint.config.js 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 GitHub Actions for continuous integration. Your latest commit on GitHub must show a green checkmark denoting a successful run of the npm test command. Be aware that this checkmark can take a few minutes to show up, as GitHub Actions runs the test.

All starter code for this class includes the .github folder, which contains configuration for the GitHub Action that will test your code. The test should run automatically for each commit. If you don't have the .github folder, you'll have to grab it from another project.

STREAMING MEDIA HW

Rubric

DESCRIPTION	SCORE	VALUE %
Assignment Completed – All steps of the assignment have been completed successfully.		**15**
Server correctly sends back pages to the browser – The client, client2 and client3 files are all successfully delivered to the browser on the correct URLs (/, /page2. /page3). The client file is the default/index page.		**15**
Party.mp4 correctly streamed back to the browser – The party.mp4 video is correctly streamed to the client. Browser can open multiple tabs and have them stream individually. Can be accessed directly at the /party.mp4 url.		**15**
Bling.mp3 file streamed back to the browser – The bling.mp3 audio file is correctly streamed to the client. Browser can open multiple tabs and have them stream individually. Can be accessed directly at the /bling.mp3 url.		**15**
Bird.mp4 file streamed back to the browser – The bird.mp4 video file is correctly streamed to the client. Browser can open multiple tabs and them stream individually. Can be accessed directly at the /bird.mp4 url.		**15**
Streaming code refactored into reusable functions – The streaming code has been refactored into functions that can be easily reused by any file/request. The party, bling and bird media files all use the new reusable code.		**25**
Heroku Penalty – If your app is not functional on Heroku, there will be a penalty.		**-10% (this time)**
GitHub Actions Penalty – If your build is not successful on GitHub Actions, there will be a penalty.		**-10% (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 2 Errors 3 Errors 4 Errors 5+ Errors	**-5%** **-10%** **-15%** **-20%** **-30%**
Missing Links – Missing appropriate links in submission.		**-10% EACH (this time)**
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 the following links in the submission comments.*** **Do NOT include your node_modules in the zip or github repo.**

- link to your Heroku App ***(not the dashboard but the working web link from 'open app')***
  link to your github repo for this assignment