1. Things for later

Important Script to run on server:

cd /opt/bitnami/projects/FullStackServer/  
forever start ./forever/development.json

To stop:

Forever stopall

All the important stuff is in /opt/bitnmai/projects/FullStackServer.

./forever/ was put in there to run the forever stuff. Development.json was created to house the start script stuff.

Big thing to note: We DON’T use “node app.js”, we use “node ./bin/www”. It has something to do with routing.

SSL certs for HTTPS are stored in /opt/bitnami/letsencrypt/certificates as marcnettles.com.crt and marcnettles.com.key

Bncert-tool used for certificates. “sudo /opt/bitnami/bncert-tool” was used with marcnettles.com and [www.marcnettles.com](http://www.marcnettles.com) as domains. http to https redirect selected, as well as non-www to www redirect.

(NOTE: Stopped apache services and everything still works… so I guess we don’t need it?)   
Apache configuration is at “/opt/bitnami/apache/conf/bitnami”, and the DocumentRoot pointed to “/opt/bitnami/apache/htdocs

2. Setting Up AWS Lightsail with Nodejs

1. Made the AWS account using Nodejs configuration
2. AWS pretty much sets up everything for the server and gives an ip address
3. Followed instructions to get static ip address
4. Followed bitnami instructions (<https://docs.bitnami.com/general/infrastructure/nodejs/get-started/get-started/>) to get the nodejs stuff started.
5. Couldn’t figure out how to get my files on this new server, so after some research:
   1. Have to create SSH connection
   2. Had to download the .pem file which contains the default key
   3. Had to download PuTTY and use PuTTYgen to import the .pem key and convert it into a .ppk key
   4. Took that .ppk key and was able to connect to the website’s public ip address with port 22
   5. Then, had to take the .ppk file and load it into a FTP browser (I used WinSCP). Then, after uploading the .ppk file, I connected to the public IP address with port 22 and was able to transfer my files over.
6. Then there were a bunch of issues starting the server with “npm start” (kept saying port 3000 is in use)
   1. Used “node app.js” instead from location “cd /opt/bitnami/projects/FullStackWebsite/”
7. Domain name is registered as marcnettles.com, but still hasn’t worked yet.

3. Accessing files by FTPing into port 22

1. Use something like WinSCP to FTP into the public ip address with port 22.

2. Click advanced and go to SSH-> Authentication and import the private key file (.ppk)

3. Click login, then use the username “bitnami” to log in.

4. Trying to fix the port problem (where site is only accessible by adding :3000 to url)

In “/opt/Bitnami/apps/letsencrypt”, there’s a “conf” directory which contains “httd-prefix.conf”. I added ProxyPass / <http://127.0.0.1:3000/> and ProxyPassReverse / <http://127.0.0.1:3000/> to it.

NOTE: This was fixed by blowing up the instance and trying again. It all seemed to work that time.

5. Upgrading XSS protection and more

Added CORS: <https://docs.bitnami.com/aws/infrastructure/nodejs/administration/enable-cors-nodejs>

But I also put it in the vhosts folder with the FullStackWebsite-http-vhost.conf file and the FullStackWebsite-https-vhost.conf file

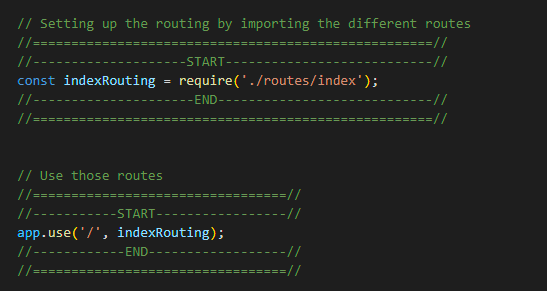
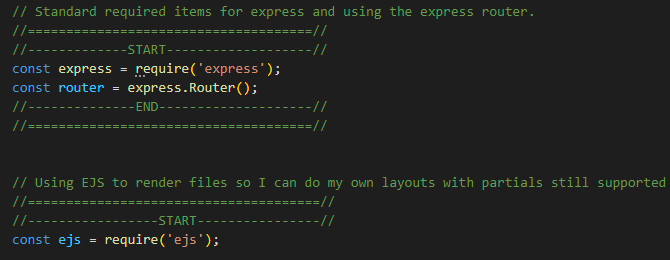
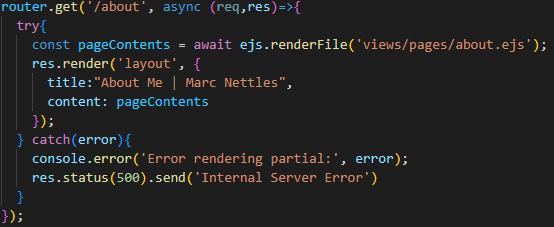
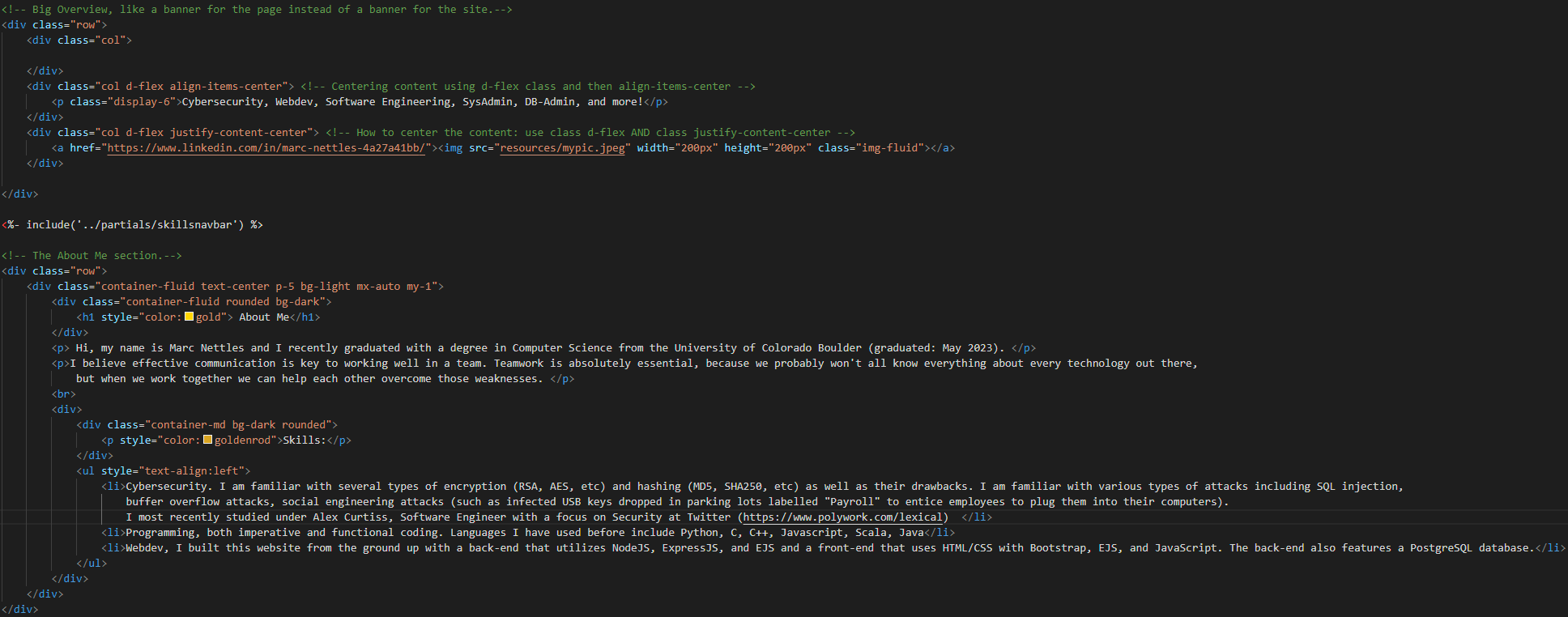
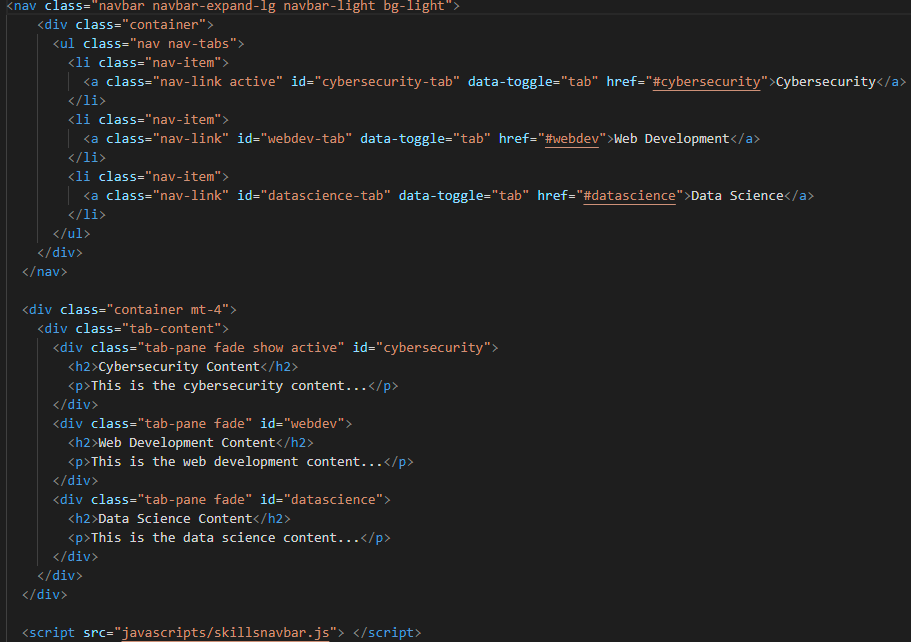
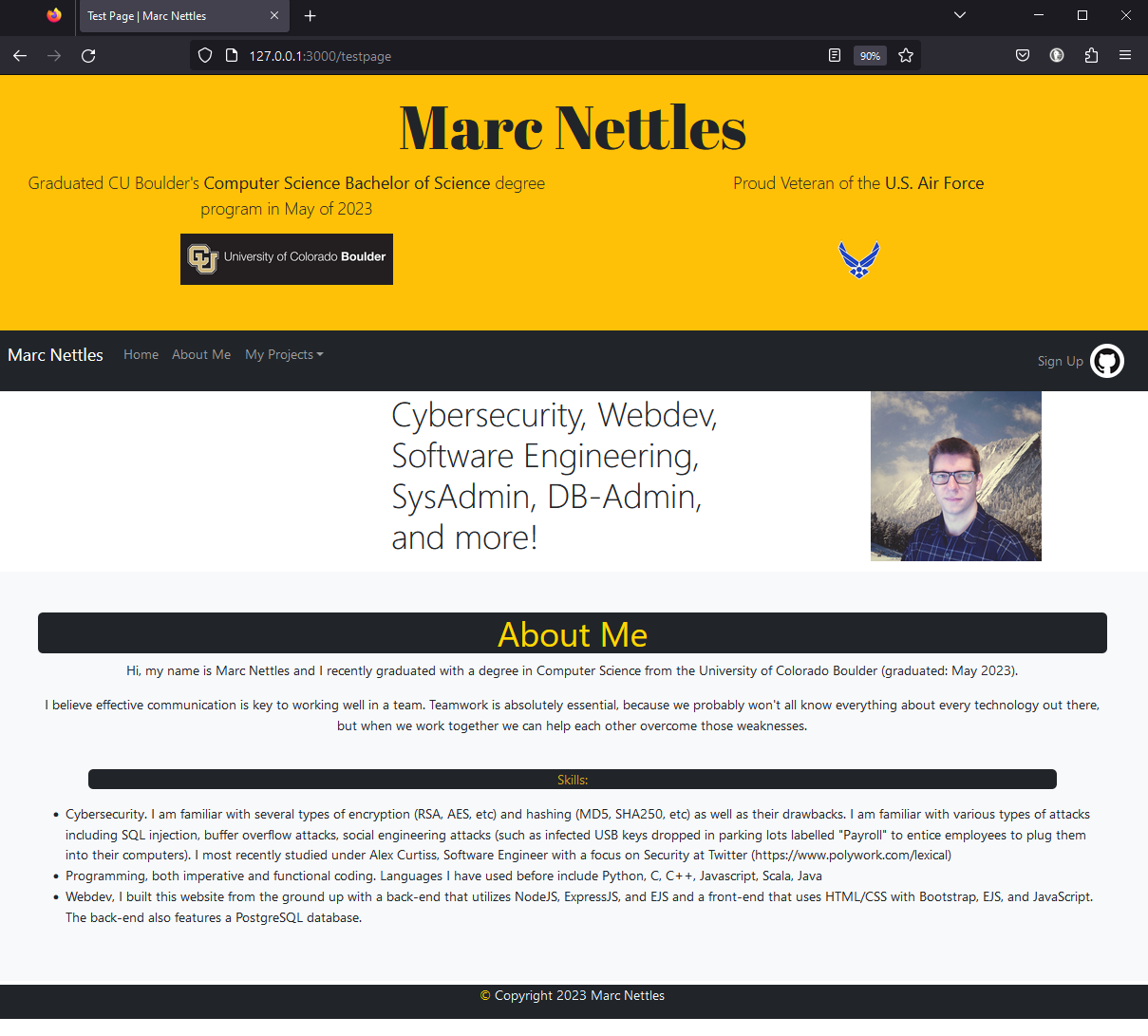
**Also added Helmet.js by doing npm install helmet.**

6. Got EJS layouts working in a weird way

**How to Inject HTML into EJS Layouts on a Nodejs/Expressjs Server (and still use partials in your pages!)**

My last post covered the same topic, but it had one big issue: EJS files loaded in with fs.readFile() no longer filled in the <%- include()%> with anything!  
  
To fix this, we need to do a couple of things.

1. Stop using fs.readFile and switch to ejs.renderFile
2. Add asynchronous support for the router.get()'s callback function. This way we can use "await" with the file render.

I'll recap what we did in the previous blog post for those that don't want to read it again.  
First, I'm routing all the traffic through the Express Router (this is in my app.js file):  
  
  
Next, in the /routes/index.js file, we grab a Router from Express and then we require('ejs') so we can use ejs.renderFile().   
  
  
Here's where things change a bit from the previous blog post, we do our routing, but we make our anonymous callback function asynchronous using the "async" keyword. This let's us use the "await" keyword when we try to use ejs.renderFile(). We also added a "try-catch" to catch any errors.  
  
  
And here is the about.ejs file, which contains an ejs directive to include a partial, skillsnavbar:  
  
  
And here is the partial:  
  
  
  
The result is the same as before:  
  
  
**And there we have it!** We have fully injectable content into layouts! And that content can have content injected into it as well. Thank you for your time!

**OLD POST**

**So, the layout.ejs file in views had me confused because it seemed like the only thing you could do (since EJS doesn’t support block layouts, which I assume means being able to replace blocks of content in the layout) was append a layout to the beginning or end of the html, which wasn’t ideal.**

**So, instead, I figured I could use Node.js with Express to fill in the data for me by having it render the layout page and then just injecting my custom blocks of html into a variable on the layout.**

**First, I’m routing using the EJS Router. Then, in the app.js, I tell the app to use this routing.**

A screenshot of a computer program

Description automatically generated

**Then, in the /routes/Index.js file, we require the fs (FileStream) to read a local html (or ejs) file. We also grab a Router from Express.**

A computer screen shot of a program

Description automatically generated

**Then we do our routing. Here’s an example of how it normally looks:**

A screen shot of a computer program

Description automatically generated

**And here’s how I change it up:**

A screen shot of a computer code

Description automatically generated

**If you’re looking at this sideways and it still isn’t making sense, let me break it down:**

* **Router.get(‘/testpage’, (req,res)=>{**
  + **This is how Express routes traffic when it receives a GET request from a browser for the page “/testpage”, like if you were to go to** [**www.marcnettles.com/testpage**](http://www.marcnettles.com/testpage) **then my server would route your GET request for “/testpage”.**
  + **The (req,res)=>{ //stuff } is what is called an “Anonymous Callback function”. Callback functions are asynchronous, meaning they won’t tie up your server trying to process this request. It pulls in any information from the request with “req” and then it uses “res” to send information back, such as rendering a page for the user to see.**
* **fs.readFile(“views/pages/aboutTest.ejs”, function callback\_read(err,data){ //stuff }**
  + **This is reading in a file (an ejs file which just contains the html for an About Me article), but it is doing it asynchronously to avoid tying up the server.**
* **If (err) { throw err; }**
  + **If the server can’t find the file, it throws an error.**
* **res.render(‘layout’, { title: “Test Page | Marc Nettles”, content: data });**
  + **This renders the layout.ejs file in our views folder and it replaces the variable “title” with “Test Page | Marc Nettles” and “content” with “data”. “data” is the output of the fs.readFile() function we saw above. The contents of layout.ejs are given below:**
  + A screen shot of a computer program

    Description automatically generated
  + **This layout is built with EJS partials (which are just pieces of HTML in a separate file that are then loaded in here), so we don’t immediately see where the variable “title” is located, but the important part is in MAIN where we see the line “<%- content -%>**
  + **<%- variableName -%> is javascripts way of embedding unescaped HTML while removing whitespace. Use <%- variableName %> if you don’t want to remove whitespace.**

**So, we now know that the variable “content” is being replaced with the variable “data” which is filled by fs.readFile() to be the contents of the file aboutTest.ejs, which is seen below.**

A screenshot of a computer program

Description automatically generated

**Which gives us the result:**

A screenshot of a computer

Description automatically generated

**And that’s it! That’s how to asynchronously inject block content into an EJS layout!**