I originally decided to create this project to experiment with new technologies and to practice with technologies I have already learned. To that end, this project was a huge success!

I was interested in using Docker to wrap my project in a container, so it is more portable and isolated from the main system it is running on. Though only creating one project and containerizing it alone felt redundant to me, I could just host one web application using NGINX and call it a day. So, I decided to create a platform for me to host a multitude of my projects in one place, and to advertise myself alongside these projects.

The first problem I needed to tackle was creating a web application that would be the face of it all. I wanted this to be visually appealing and easy to navigate after all this is the first thing any visitors to the site will see. For this piece, I decided to use the popular React library for JavaScript. I love using to create the bones of a site instead of just using HTML. Being able to create reusable components instead of just rewriting the same HTML repeatedly significantly speeds up development time and makes the overall code base smaller.

For example, in this project, I wanted to be able to add new projects or maybe delete old ones with ease. Using React I was able to create a project component that accepts its information as props.

```
<div className="project">
  <a className="project-link" href={link} target="_blank" rel="noopener noreferrer">
    <img className="project-image" src={props.img} alt={`Screenshot of ${props.title}`} />
  </a>
  <div className="project-details">
    <div className="project-tile">
      <p className="icons">
        {props.techIcon.split(' ').map(t => (
          <i className={techIcons[t]} key={t} />
        ))}
      </p>
      {props.title}{' '}
    </div>
    {props.children}
    <div className="buttons">
      <a href={repo} target="_blank" rel="noopener noreferrer">
        View source <i className="fas fa-external-link-alt" />
      </a>
      <a href={link} target="_blank" rel="noopener noreferrer">
        Try it Live <i className="fas fa-eternal-link-alt" />
      </a>
    </div>
  </div>
</div>
```

The above picture contains the markup for one project element. It should also be noted that this is slightly different than what true HTML would look like because of things like props and the use of JavaScript methods inside tags. If I wanted to create

two, three, or more of these I would have to replicate this markup and more for each one.

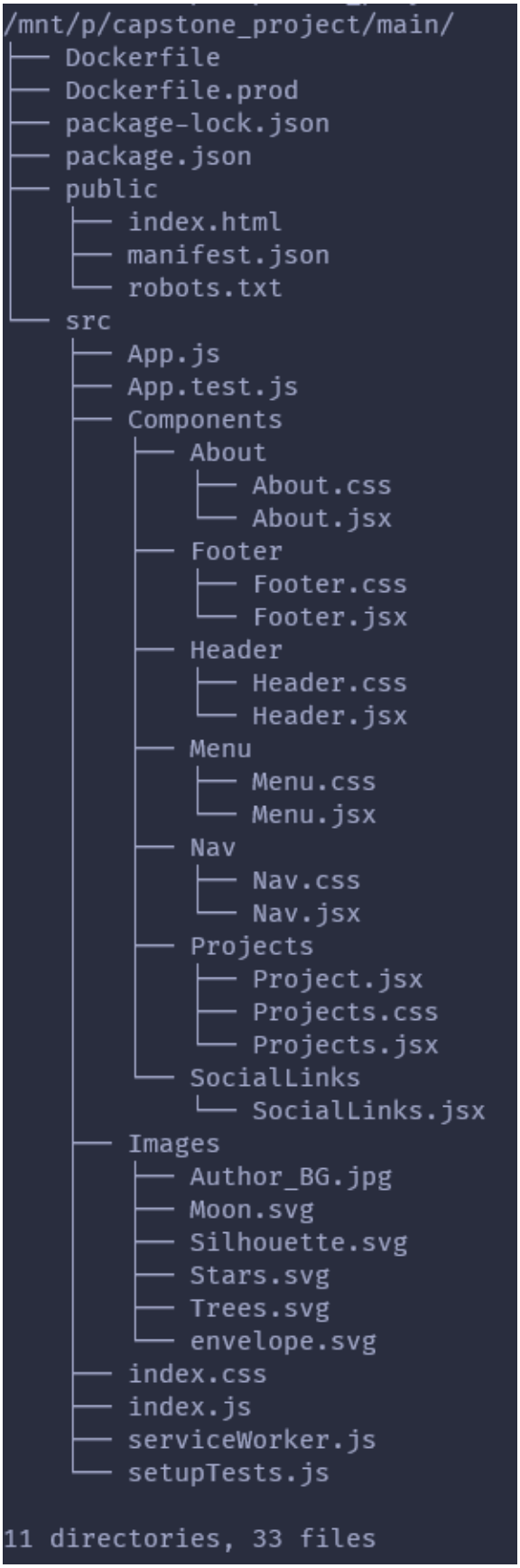
Next, in the below picture we can see what must be passed to our React component to have it create an example project element. As we can see it is less code, and easier to read. And creating more of these is as simple as replicating what you see below and changing out the information to reflect the project being shown.

```
<Project
  title="Example Project"
  img={''}
  techIcon="js css react node sass"
  link={null}
  repo={null}
>
  <small>Built using Node, CSS, and React.</small>
  <p>
  There isn't actually a project here, this is just an example
  of what could be! This being the case none of the buttons will take
  you anywere.
  </p>
</Project>
```

Still, all the components I created would not be nearly as visually appealing without the use of CSS. Cascading Style Sheets or rather CSS is an absolute necessity for stylizing a web application and for making sure it is responsive.

Responsiveness is another problem I focused on a lot when stylizing this project, I wanted it to look just as good on mobile as it does on a desktop. To make it responsive I had to set different values for certain attributes such as height, width, and positioning based on the screen size. All of this can be done through CSS and generally just requires some trial and error to find what settings work best for different screen sizes.

An interesting problem arose that I had not thought about until this project, this being the compatibility of CSS in different browsers. I had someone else view my project to give me their thoughts and they said a style was completely broken, though when I looked at it was exactly as intended. As it turned out the way I was handling opacity was not supported in the android version of Firefox, but it works as intended on the desktop version of Firefox and Chrome on both mobile and desktop. This and a couple of other issues due to compatibility popped up throughout the styling process, so for several things I had to create redundant styles in case one or more options were not supported. For the opacity fix, I had to normalize the percentages I was using, and I added a rgba opacity of the same value for extra compatibility.



To keep things organized each component has its dedicated CSS file, and inside the CSS files, I worked to keep tags in the order as they appeared in the JSX file and to keep attributes in alphabetical order.

Speaking of organization, that was something I wanted to focus on in this project as well. I tried to keep each file orderly and make the code easy to read, though sometimes I failed, or I could have improved it more. As for the file structure of the project, I feel like I found a good method of organization. Using the create-react-app tool creates an src folder which is generally where people store their components. I decided to go a step further and add a file for Components, and a file for Images separately. Then inside components, there is a folder for each component. Or for related components as is the case for Projects. Each folder is capitalized and is named the same as the component it contains. Inside each are the CSS and the JSX file to its respective component.

The adjacent photo is a tree diagram of the file structure for this project. I am excluding the folder for the node_modules as this would cause a significant number of files to be shown that is not relative to this project. This is also why I included node_modules in the .gitignore file and the .dockerignore file for this project, that way it will not include this bulky directory in the GitHub repository, or when I create the Docker images.

Once all of this was created it was time to implement Docker! Docker is a technology I had never worked with before, so I had to do a lot of research on how to use it and best practices. Their sales pitch explains why I thought it would be perfect for this project. “Developing apps today requires so much

more than writing code. Multiple languages, frameworks, architectures, and discontinuous interfaces between tools for each lifecycle stage creates enormous complexity. Docker simplifies and accelerates your workflow, while giving developers the freedom to innovate with their choice of tools, application stacks, and deployment environments for each project" (docker.com/why-docker). I do not know what types of projects I will want to develop in the future, what languages I will use, what frameworks, or anything else. Having the ability to support any future endeavor I may take on made Docker perfect for my needs.

Docker enables the use of what it calls a container. These containers work similarly to a very lightweight and efficient virtual machine. Containers are created from an image which is an executable package of all the software, runtime, system tools, system libraries, settings, and configuration files need to run an application. They are incredibly flexible, expandable, efficient, and fast.

Using Docker containers, I could isolate each project I make, including this capstone project from everything else. Each container would have all the files and dependencies to make its respective project work and will not interfere with the others in any way.

```
root@shell:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
357a8ff4bd42        deliverables:prod   "nginx -g 'daemon of…"   24 hours ago        Up 24 hours         0.0.0.0:8081→80/tcp   deliverables
01e688fd3c64        portfolio:prod      "nginx -g 'daemon of…"   5 days ago          Up 2 days           0.0.0.0:8080→80/tcp   portfolio
```

In the above picture, we can see that I currently have two containers running. One is for the deliverables piece of my capstone, and the other is for the main project. Both projects are running on my Vultr Virtual Private Server (VPS), yet neither know they are on a VPS or of the other's existence. They are completely isolated and independent.

I still wanted them both to be hosted through my one singular domain. To do this I used NGINX running on the VPS and a reverse proxy.

```
server {
        listen 443 ssl;

        server_name lapagejenkins.dev www.lapagejenkins.dev;
        ssl_certificate /etc/letsencrypt/live/lapagejenkins.dev/fullchain.pem; # managed by Certbot
        ssl_certificate_key /etc/letsencrypt/live/lapagejenkins.dev/privkey.pem; # managed by Certbot


        location / {
                proxy_pass http://localhost:8080;
        }

}

server {
        listen 443 ssl;

        server_name capstone.lapagejenkins.dev;
        ssl_certificate /etc/letsencrypt/live/capstone.lapagejenkins.dev/fullchain.pem; # managed by Certbot
        ssl_certificate_key /etc/letsencrypt/live/capstone.lapagejenkins.dev/privkey.pem; # managed by Certbot

        location / {
                proxy_pass http://localhost:8081;
        }
}
```

The first section labeled the server is for the portfolio docker container, while the second is for the deliverable's container. Both containers have their port 80 exposed, and when running them I map the VPS' port 8080 to portfolio port 80, and 8081 to deliverables port 80. If this was on my computer I could go to http://localhost:8080 or http://localhost:8081 to view the website hosted by each container.

Using NGINX and reverse proxy I mapped my domain lapagejenkins.dev and the www.lapagejenkins.dev subdomain to the http://localhost:8080f for the portfolio container. Then I added a DNS record for the subdomain capstone.lapagejenkins.dev and mapped it to http://localhost:8081. I also had to acquire SSL certificates for all these domains to ensure HTTPS was available. I did this by using the certbot utility. Once all of this was set up both sites were publicly available on my domain. Again, my domain name lapagejenkins.dev plus the subdomain www.lapagejenkins.dev is hosting the portfolio/main project site, while the subdomain capstone.lapagejenkins.dev is hosting the capstone deliverables site.

With everything in place, the overly simplified steps to add a new project go as follows:

1. Create the project
2. Wrap the project in Docker container i.e. containerize it
3. Clone the repository for the project onto my VPS
4. Build the docker image by running: docker build -f path_to_Dockerfile -t sometag:dev_or_production root_directory_of_project/ --rm
5. Run the image with docker run -d --restart always -p system_port_#:container_port_# --name container_name the_tag_created_when_building_image
6. Add a DNS record for a new subdomain
7. Add a reverse proxy to my NGINX configuration
8. Use certbot to generate a new SSL certificate for the subdomain
9. Restart NGINX
10. View the project in the browser

For updating an existing project and its container, I do as follows:

1. Cd into the directory for the project
2. Get the updates from GitHub by running: git pull
3. Run: docker stop container_name
4. Run: docker rm container_name
5. Run docker build -f path_to_Dockerfile -t sometag:dev_or_production root_directory_of_project/ --rm
6. docker run -d --restart always -p system_port_#:container_port_# --name container_name the_tag_created_when_building_image

I will note that the docker build and docker run commands may need different options, parameters, and arguments based on the needs for that project. All these options are well documented in the Docker documentation.

Another side note is that you could publish the images to the Docker hub, which is like creating a repository on GitHub. In that case, you could skip the steps related to GitHub and the docker build steps, instead use the tag from the Docker hub when running docker run.

I would have liked to put NGINX itself into a docker container and use the docker proxy tool to create a network just for the containers. By doing that all the web sites and the server would be isolated from my VPS. Though I did not have the time to keep researching this topic for a suitable method of doing so. This was mostly due to the issue of requiring SSL certificates for HTTPS, and HTTPS is required when using a domain ending in .dev.

For what I wanted to learn, experiment with, and practice this project was a huge success. I learned a lot about Docker and how to use their containers to isolate my applications. I was able to practice using React and learn more about best practices, and the component lifecycle. I realized the difficulty in creating a responsive design that is compatible with the growing number of web browsers and the outdated versions of said browsers. This was the first time I had used Vultr for a VPS and NGINX for a web server. Vultr was incredibly easy to use, and even includes an image that comes with Docker preinstalled on Ubuntu 18.04. NGINX made creating the reverse proxies as simple as adding a few lines to a file, without this, I would not have been able to host multiple containers from one domain the way I did. Now I look forward to adding many more projects to my VPS, and my site.