Senior Project in Computer Science

Capture The Flag (CTF)

Cybersecurity

Alex Miller

Jordan Latimer

Project Advisor: Dr. Michael Kowalczyk

Special Thanks To: Dr. Jim Marquardson

Winter 2025

Introduction

The idea for the project started when we took CS495: Software Engineering with Dr. Kowalczyk during the Fall semester of 2024. Dr. Marquardson, a professor of cybersecurity here on campus, had reached out to Dr. Kowalczyk, seeing if there was any interest from his students in taking up a personal project he'd like to see come to fruition. The project was a CTF (Capture the Flag) implementation to be used as an academic tool for his students. In cybersecurity, CTF is a training exercise, or in some cases a competition, where the goal is to gain access to a hidden string of text. The text (or flag) is often encoded as "FLAG{FLAG_NAME}", and upon successful retrieval of the flag, the user's score will increment to reflect that. The challenges in these exercises can vary in difficulty immensely, which makes it great for users of all skill levels. For example, a beginner level flag could be to test the user's ability of basic linux commands such as cat to display a file's contents to reveal the flag. The challenges can get much harder, and the only constraint is the flag creator's creativity.

This proposed project intrigued us both, so we took up the offer to work on it for our semester project. After taking up the project, Dr. Marquardson had told us the reason why he'd like to see this project become a reality in the first place. For the hands-on CTF-style problems his students typically used, it would take 5-10 minutes to launch a hands-on challenge, and at which point every flag would be the same for all users. Now, this is obviously very inefficient for use in a classroom setting, where there are time constraints and no real means of preventing cheating. We wanted to develop a system that solved both of these issues. For the unnecessary long launch time on flags, we opted for the use of Docker containers to hold onto the environment, as they are very lightweight with minimal startup time. And as for unique flag hashes, the solution for this was instead of having a static flag for the challenge, to hash the flag based on the user's email and that particular flag’s ID in the database, ensuring dynamic flags.

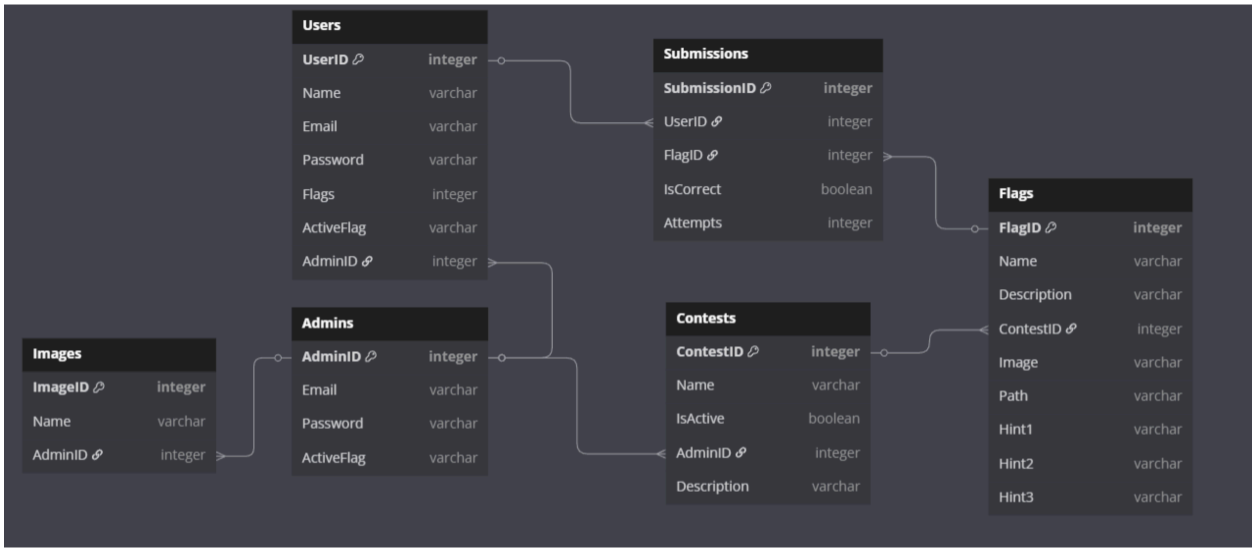
By semester's end we had implemented only part of what we had set out for, due to many reasons. There was some basic user interface, a SQLite database to hold data and query to, and a system to place a 'flag' at a specific file location, of which would randomize upon page reload. There was also frequent crashing that had yet to be fixed. After the fall semester ended, we both knew we had to do our senior projects the following winter semester, and decided it would be a great opportunity to hopefully finish what we started. Dr. Kowalczyk agreed to let us do it, and Dr. Marquardson agreed to continue being a sort of 'consultant' role.

Changes from CS495 to CS480

There were a lot of changes from where our project was for CS495 compared to now for our senior project. The scope of this project grew, not only because it’s a senior project but also because we realized the potential of this project and where it can go. This semester, the first course of action was to restructure our current code. The way it was organized and written wasn’t as good as it could have been. All the files were in the same directory, the server code wasn’t organized properly, a lot of our functionality was missing, and we had several crashes. Organizing our code also helped with scalability which was a huge part of this semester as we increased the scope. After that, it was fixing what already didn’t work and what we didn’t get the time to add during our time in CS495. Then it was about scaling and adding more features and functionality to the project that we wanted to add. Some big changes made were the switch from SQLite to MySQL and the addition of Angular.js. Even the way we dealt with the containers was made more optimized such as checking if the container exists already before creating it along with lots of error checking everywhere. Security was an issue we didn’t tackle so that was also a focal point in our scope this semester. Everything that we wanted to change is in our proposal and we were not able to get to all of them plus some were scrapped such as: system stats for Admins as we don’t believe it’s needed. Before we got into any of the new changes for this semester's scope, we had to do some cleaning up so the first portion of this semester was dedicated to that which was a fantastic idea in the long run.

Roles Interactions / Database

Storing all the different sorts of data such as contests, admins, and users effectively was going to be an integral part of getting this project off the ground. In the beginning, before we decided to overhaul this project, we had decided to go with SQLite, as it was lightweight and didn't require a server. After the decision to expand the scope of this project, the need for a more scalable database system became more obvious. This is where MySQL comes in, it's much more suited to handle larger datasets and is designed for concurrent access by multiple users and applications. Every page reload makes one or more database queries, not to mention all the calls in the other standard usage of the application. Below is a relationship diagram for the tables used in our design.



The most important interactions in this design come from the role of the admin. They are responsible for creating students, contests, flags within those contests, and images for the terminal environment. Another notable relationship

Keeping Sensitive Database Data Secure

For the first good chunk of development, we were just keeping the user passwords in the database as plain text for debugging purposes. This obviously isn't very secure at all, nor is it a good software development practice. So we decided to encrypt the passwords using the "bcrypt" package. Bcrypt is a time tested encryption method that has been around for many years. What it does is take in a "salt round" variable, which basically translates to its cost factor. The cost factor controls how much time is needed to calculate a single bcrypt hash. The higher the cost factor, the harder it is to crack, but at the same time, the more time the application needs to compute it. A good trade standard is a salt round of around 10-12, we decided on an even 10 to ensure app efficiency.

Flags

Creating unique flags for each user was a focal point from the start, and something we weren't able to fully achieve last semester. This was addressed by incorporating the user's email into the flag hash. The hash is created by encrypting the email with that particular flag's ID using a SHA-256 hash function. The first 8 characters of the resulting hash get placed into the flag, which in our format is "NMUCTF${ [hash] }". When a user selects a flag in the contest menu, a container is created from the flag's associated image (we go more into this later). At the same time, the flag gets placed at the flag's specified path using the line:

```
const exec = await container.exec({
            Cmd: ['/bin/bash', '-c', `sed -i 's|\\[FLAG\\]|${FlagHash}|g'
${flag.Path}`],
        });
```

The 'sed -i' command replaces all instances of a given string in the file. [FLAG] is the identifier we use to find where the flag should be placed. When a user finds the flag and submits it, a check is done on the backend to ensure that it is correct. The Submissions table in the database holds on to the amount of attempts it took a specific user to get the correct flag, or if they got it at all. The data from the Submissions table is used in multiple ways, such as the leaderboard that shows who has collected the most flags.

FrontEnd

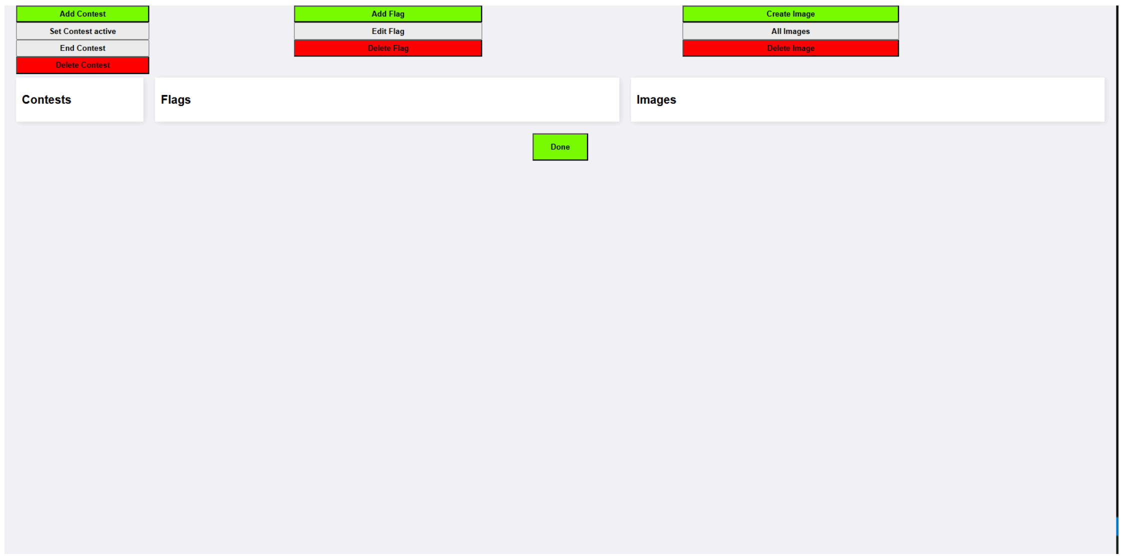
Similarly to the decision on what database to use, as the scope of the project changed, so did our decision to implement a front end framework. Initially, there were no plans to use a framework, but as the code base expanded, it got much harder to maintain and organize. We recognized that significant refactoring was needed. While neither of us had much prior experience with Angular coming into this, we were aware of some of the features and built-in tools it had to offer, one of which was route guards for authentication. We were eager to learn more about the ins and outs of Angular, knowing this knowledge would be valuable long-term. It was around this time that we were heavily focusing on improving the application's security, so Angular's built-in AuthGuards made it an even more compelling choice.

Angular Perks

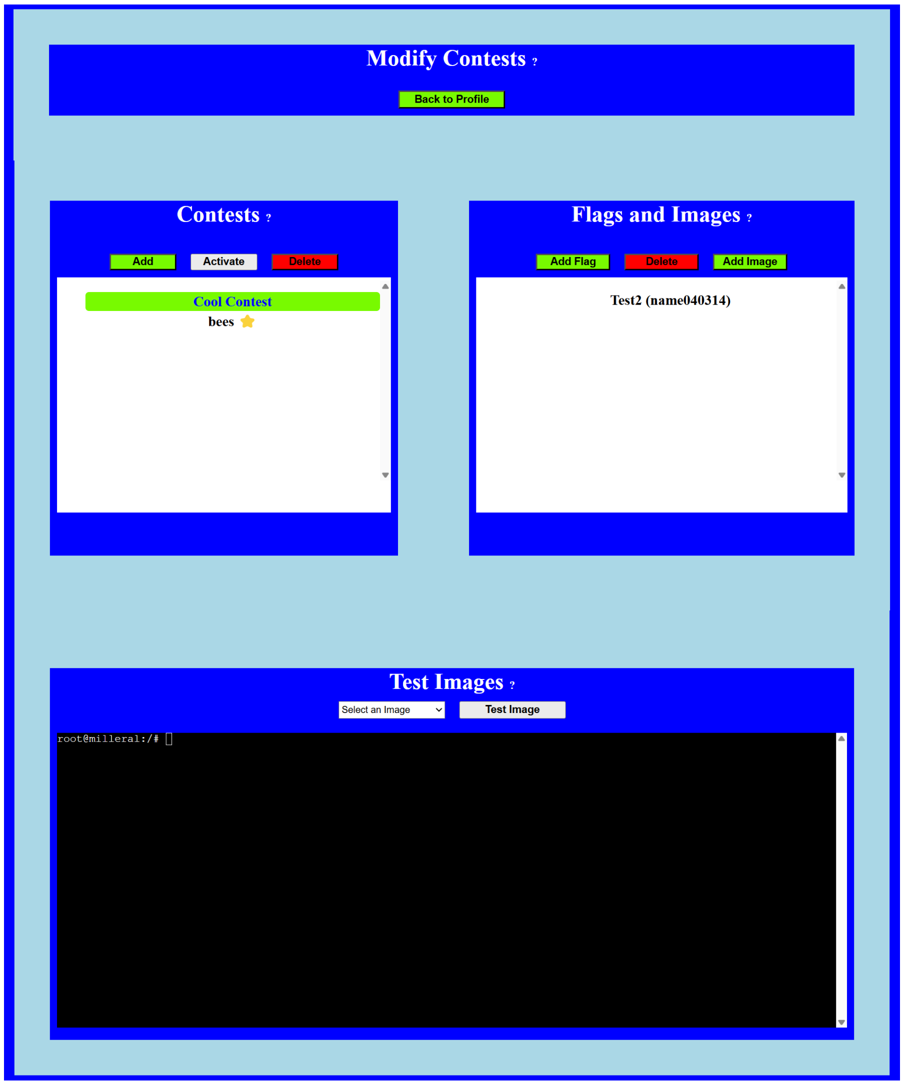
As we learned more about Angular, the decision to make the switch made even more sense. One aspect we had neglected up until pretty late in the development cycle was security. Thankfully, Angular has built-in security measures called AuthGuards. What you can do is set up a check so whenever a user tries to access a page, the routing component set up with AuthGuards will make sure they are authorized to do so. If they aren't, they can be routed to a separate page like a designated 'unauthorized' component, or just back to the login screen .We also incorporated session storage into the app, which holds onto certain information such as roles or a selected contest's ID. This data is stored until the user closes out of the tab, and persists between page loads. This worked out perfectly, as before we were putting this information into the URL of the page itself, and using functions to retrieve that data. Now, that data could just be grabbed from session storage. This worked great, up until maybe a week later where after some research we realized sessionStorage was not secure whatsoever. After that we pivoted to using sessions created by the backend itself, using a time-out of a specified time period to release the stored session data. This is a much more secure route to go, although we did continue using AuthGuards for less sensitive information. Switching to angular also helped make the project much more organized. Instead of having a list of all the files, we could break them down into components. Each component creates a css, html, and typescript file for a given component (or page).

User Interface

Creating the UI was more of a challenge than we had originally thought. CSS isn't either of our strong suits and neither of us have ever been artistically inclined. Even so, we didn't want to just implement a random bootstrap and call it a day. We set out to create a unique user experience instead, so all of the styling was done manually. A lot of the UI relied on a grid layout so we can put each division element in the correct spot. UI for this project wasn't just on the CSS but also the wording we used, where things were located, and if the overall page made sense to the user. User experience played a big role in how we did the UI. In a meeting with Dr. Marquardson, he pointed out some things that just didn't make sense at the time and how the overall design was awkward and confusing for several pages. Getting an outsider's perspective really helped us nail down how all the pages should look. Here is an example of how our old Modify Contest Page used to look (with no data inserted).



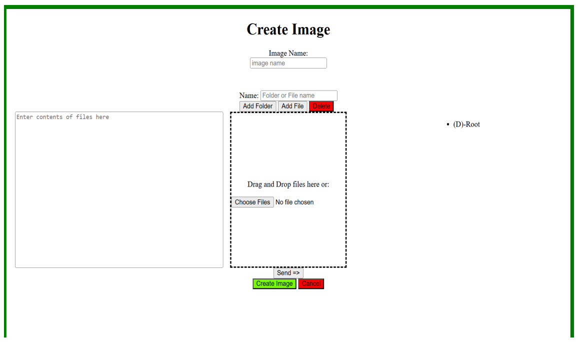
You can just tell that it wasn't the greatest. With so many buttons across three different sections, it was tough to pinpoint the relationship between Contests, Flags, and Images. With the placements of the buttons as well, it was in a really awkward spot. Even the button at the bottom with the dynamic changes to the lists if there was enough content of just one of the lists, the button would also move down making the user scroll all the way down just to exit. With the new and improved way below, we addressed many of these issues.

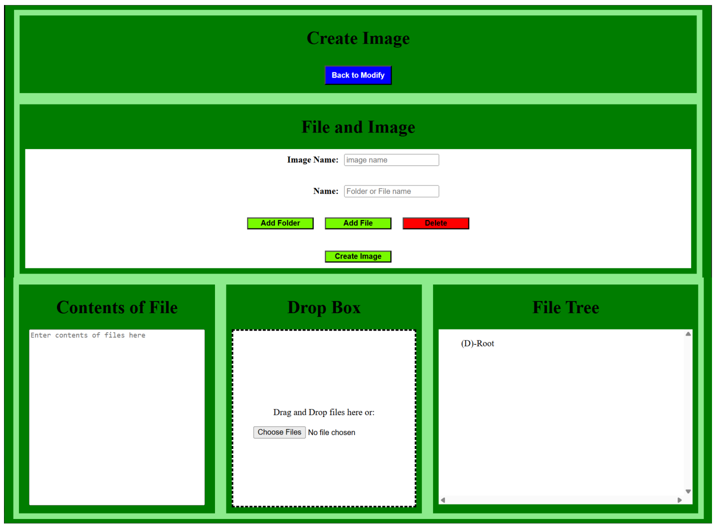


```
root@milleral:/# 
```

You can really see the massive improvements, it might be weird looking at this page considering everything else in our project is green, this is done by design. Modify Contests is blue for one main reason, the Admin. With it being blue, the Admin gets a more “settings” feel from this page compared to the others since this is where you change everything that has to do with contests. With the new design, every part has its own spot. We even added a request from Dr. Marquardson to let the Admin be able to test out the newly created images before you even have to assign it to a flag thus showing the relationship between flags and images even more. We also added “?” to each section and when it's hovered over, it displays a little text box showing what that section is and how to use it. In the Contests section it is easily indicated which one is selected with it highlighted in green while the star represents which contest is currently activated. Speaking of activation, we parted ways with having two buttons for activating/deactivating a contest. It is now reduced to a single button which deactivates a contest if there is a contest active and activates the contest the Admin selected. The buttons themselves are color coded as well. Bright green showing that it's an addition and red showing its going to be a deletion.

The sectionalization of the pages is the whole bread and butter of our UI throughout this project. It keeps specific parts of the webpage separate and doesn't overwhelm the user with so much going on at once. It's a lot cleaner looking and structured which helps with scalability in the future. You can see the same thing with the Create Image Page. We had the same old look and added the new UI and it really blew it up more and made it more structured and neat.





BackEnd

Node.js

For our main backend, we decided to go ahead and use Node.js. The cool thing about Node is that we are able to import packages and use them really easily, to be able to run and stop the server really easily, and also it's widely used in development projects so there's a lot of documentation about it. Everything that we needed was inside an npm package that we could easily install and use. With the npm modules we are able to put them into the .gitignore file so that way, we don't have to install all the packages and move them back and forth when we push and pull from Git. This makes our lives a lot easier just using "npm install" whenever a new package is used.

Containers

Docker

Docker is an application where you can make "containers" that are basically mini versions of a virtual machine. They work great with what we are doing due to the fact that they are very lightweight. A base Linux is around 4GB while a container for our use averages around a few

MBs. It all depends on what exactly is inside the environment that the container is holding but either way, it is significantly smaller than base Linux. Docker has an easy way to manipulate these containers as well, an npm package called Dockerode allows us to create/destroy/execute containers very easily. Here’s a code snippet of the creation of the container.

```
// Create and Start the container with correct image
async function StartContainer(image, username, email) {

    // create container
    try {
        const container = await docker.createContainer({
            Image: image,
            Cmd: ['/bin/bash'],
            AttachStdin: true,
            AttachStdout: true,
            AttachStderr: true,
            StdinOnce: false,
            OpenStdin: true,
            Tty: true,
            Detach: false,
            Hostname: username,
            name: username
        });

        // start container
        console.log('starting container', container.id);
        await container.start();
        console.log('+++++ Container started for ' + username + ' with ID: ' +
container.id + ' +++++');
        return container;
    } catch (err) {
        // confliction or if container doesn't exist
        if (err.statusCode === 409 || err.statusCode === 404) {
            return CheckContainer(email);
        }
        else console.error(err);
    }
}
```

As you can see, There are a lot of settings that go into a container. Most notable are the “Attach” settings. These allow us to connect the websocket and allow stdin and stdout with the containers terminal. The “Image” is the image of the environment that was created (more on that below). We chose to make the functions for the container async since then we can use the keyword “await” to make sure the container is created and started. The “Hostname” and “name” is just the name that appears inside the terminal as well as the name of the container in Docker. We use these since it adds more personality to the terminal and allows us to see which user currently has a container running. The “Cmd” is the command run inside the container once it starts up, “/bin/bash” is a Linux terminal.

The basics of this function is to create a container with what is needed and send that back to the websocket OnConnection so that the container can be used with the websocket to send input and output from the front end to the container. Meanwhile the CheckContainer() just checks if a container of the name is already created and if it is, destroy it and create a new one. Whenever a user closes the websocket the container is then destroyed thus fulfilling the lifecycle of the container.

Getting the containers to work not only with the websocket but in our back end, was the hardest part of the whole project. We had numerous issues dealing with docker but once we understood it and got it working, it became a lot easier to do the rest of the project. Some issues we had was getting the docker container to connect to the websocket, being able to send through the websocket, docker has overhead with every input/output, and even creating/deleting the container we had issues with. We had to do a lot of error checking along with making the functions async to use the “await” keyword to make sure what we were doing had time to do. Doing that fixed some issues but not all of them, the error checking and the try-catch blocks when dealing with containers made sure that if we had an error, it would be handled with.

Images

This is what the environment of the container really is. Images are created from a .dockerfile which inside the file are specific commands to set up the container. The way we set up images is below:

```
FROM ubuntu:latest
WORKDIR /.
CMD [“bin/bash”]
COPY Root /.
```

Docker has their own file extension when dealing with images “.dockerfile”. With that, comes its own commands. We set up a basic Linux environment to start with the FROM command. We add what the working directory is along with the command that starts the container itself. The COPY command copies all the folders and files in a Root directory that’s created by us when an Admin creates their own image.

Recreating the whole file tree the Admin creates is not something we wanted to do as it takes up storage space on us. We even tried this same strategy but to delete the files after the image was created. That also did not work as it was too fast at deleting the files before the COPY command in the dockerfile executed. Even with await, it was still too fast so we had to go with creating the whole file tree that the Admin created in an AdminImages folder. In this folder, each Admin has a folder that is called depending on their ID number in the database. When the Admin creates an image, the tree they create gets recreated and the dockerfile gets created to make it into an image. The next time that admin creates an image, the old file system they made gets deleted along with the dockerfile ensuring that only one copy of a file system per Admin is on our system. It's not ideal but it works for what we are doing.

In the Modify Contest page, an Admin can create their own images which end up being connected to one or multiple flags. The Create Image Page deals with all of the images, an image of it is in the UI section of this paper and described below.

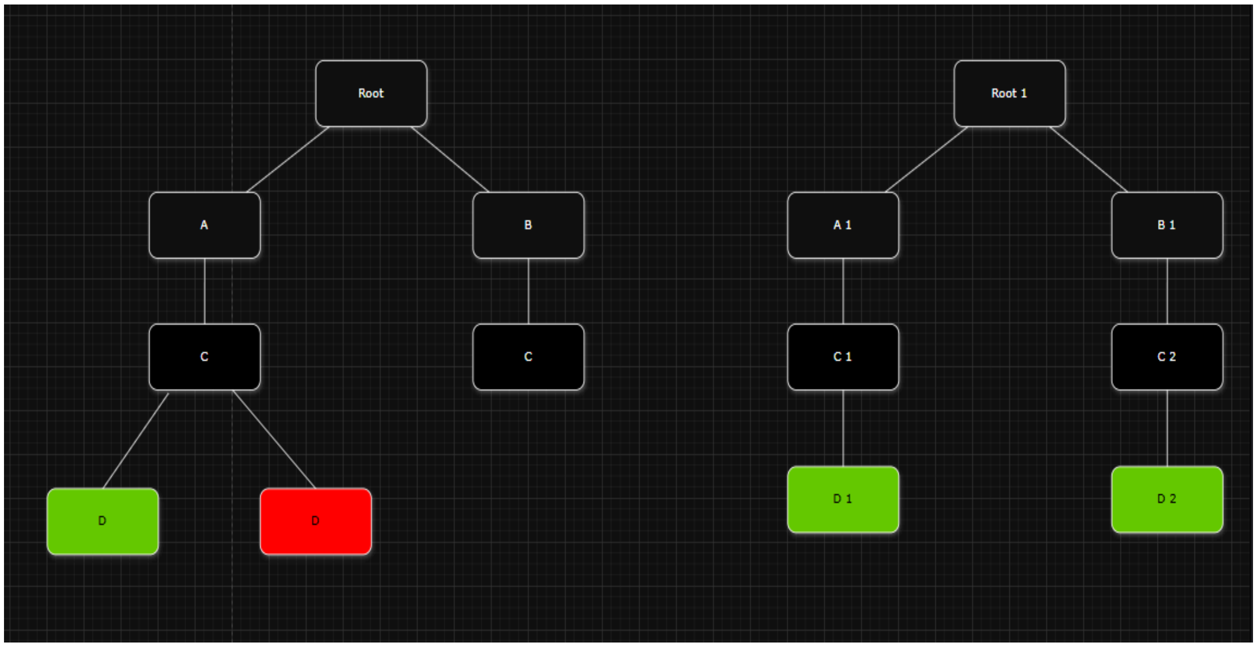
Left Side:
This is where the Admin can enter text when creating a new file. If an Admin wants to create some kind of file and doesn't have it on their machine, this is a way to create them.

Center:
Here, there are multiple things going on, the main of which is the dashed box. This is a spot where the Admin can drop files and add them to the tree. Doing it this way saves time on the Admin from having to create every file they want to add on the left side and above. It also saves any metadata that is on the file to add even more depth to what kind of flags the Admin can create.

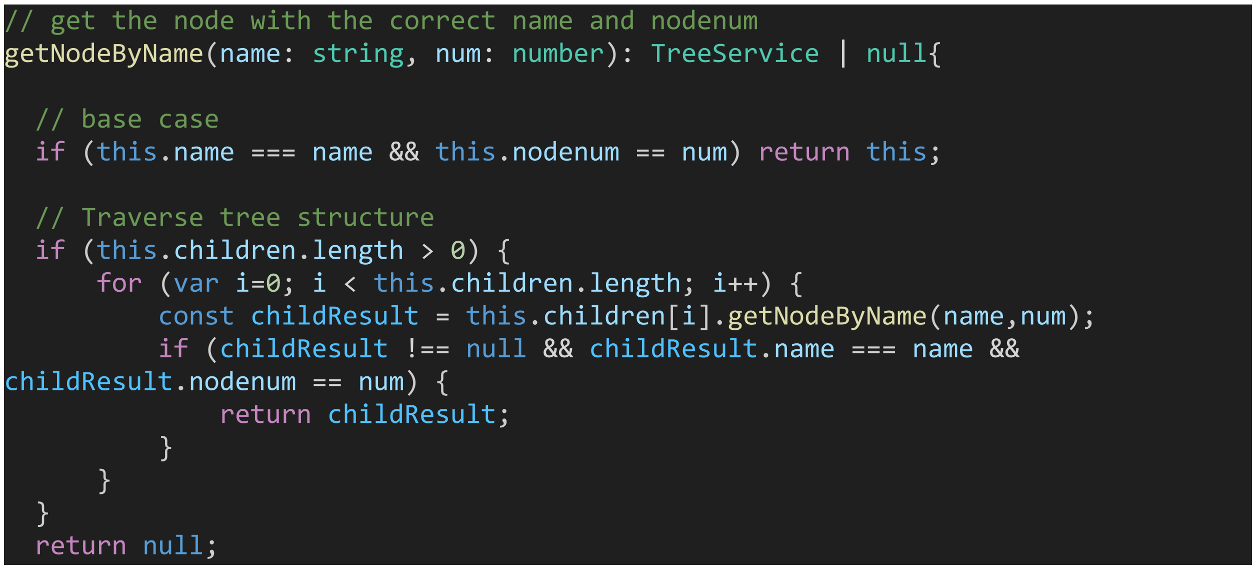
Right Side:
On this side is the tree itself which is the most interesting part of this page along with the containers in general. It starts at root and as the Admin adds and deletes files and directories it shows up here. It's a simple tree structure with each node having an array of children and a reference to its parent node. Each node also comes with a name, contents, directory boolean, and an ID number. The ID number solved a problem we didn't intend and we will circle back to that.

The biggest hurdle was sending the files themselves along with the tree structure to the server side. This was done using a FormData kind of post request. Using this, we can send an array of all the files along with the tree structure at the same time without needing to do more than one post request. When you select on a directory, it will highlight so you can see which one you selected, you can also select multiple at once allowing you to easily cascade and make multiple directories or add the same file to many at once.

The ID number fixed a really weird issue. How we designed it, when the Admin selects a directory to add we grab the list item that was selected. That way we can take the text content of that list item and find the specific node in the tree structure to make a new child from. The way we found the specific node was through a depth first search using recursion. Only issue was that in a file system you can have multiple files and directories named the same thing but be in two different directories themselves. This is the issue with depth first search is that we would find the first node that was selected, then go to find the second starting at the root of the tree again. If those two nodes have the same name, we find the same node both times, this is the issue. How we solved the issue was to give each node an ID number. The way the ID number worked was when you add a node to the tree, it counts up all the nodes that have the same name as you. Take the length of that list and add 1, that's your ID number that was then put into the class list of the list item once it was created. This way, when we look for a specific node the Admin clicked on, we get the name and what number it was added. Below is a diagram showing the issue along with the solution.



In this image, we have folders A and B, each having a folder C. When we go to add the folder D to both C folders, we start at the Root and work our way down to A then to C and add D. When we go to add the second C folder, we start at the Root and work our way down getting the same C folder that we were just at basically adding two D folders into the same C folder. With the new approach to ID numbers, we can traverse the tree looking for specific folders even if they have the same name. Instead of going from Root to A to C we now go from Root 1 to B 1 to C 2. This way makes it much cleaner and easier with looking for the correct C folder to add files/folders into. Below, is the function that traverses the tree looking for the correct node.



```
// get the node with the correct name and nodenum
getNodeByName(name: string, num: number): TreeService | null{

  // base case
  if (this.name === name && this.nodenum == num) return this;

  // Traverse tree structure
  if (this.children.length > 0) {
      for (var i=0; i < this.children.length; i++) {
          const childResult = this.children[i].getNodeByName(name,num);
          if (childResult !== null && childResult.name === name &&
childResult.nodenum == num) {
              return childResult;
          }
      }
  }
  return null;
}
```

Terminal

The Terminal on the Contest pages is directly linked to the Docker container containing the flag environment the user is currently on. It's a simple websocket connected to our Xtermjs frontend and our server backend that holds the container. Using websockets allows us to have easy communication from the client and the server by just piping what the container spits out to the user and whatever the user enters goes straight into the container's terminal. The container hijack option is used to take over the terminal allowing us to be able to pipe data in and out of it, code is shown below along with the many errors and error checking that comes with docker.

```
// take over the container via terminal in contest pages
exec.start({ hijack: true, stdin: true, stdout: true, stderr: true}, (err,
stream) => {

      // used to get rid of header from docker buffer
      const cleanser = new DockerStreamCleanser();

      if (err) {
            return ws.send('Error: ' + err.message);
      }

      // docker sending data
      stream.pipe(cleanser).on('data', (data) => {
            ws.send(data.toString());
      });

      stream.on('error', (err) => {
            console.log('stream error: ' + err);
      });

      // from terminal to docker container
      ws.on('message', (message) => {
            stream.write(message.toString());
      });

      // connection closes
      ws.on('close', async () => {
            try { // try deleting container
                  clearActiveFlag(messagestring);
                  console.log('Deleting Container: ' + container.id);
                  await container.kill();
                  await container.remove({ force: true });
                  console.log('----- Container: ' + container.id + ' has been
deleted -----');
            } catch (err) { // error catching
                  if (err.statusCode === 409) {
                        const cont = docker.getContainer(container.id);
                        cont.start();
                        cont.kill();
                        cont.remove({ force: true });
                  }
                  else if (err.statusCode === 404) console.log('404');
                  else console.error(err);
            }
      });
});
```

To remove the container itself, we use container.kill() which stops the container by force and then using the remove() deletes the container giving us dynamic creation/deletion of containers. Error code 409 indicates that the container exists but is not running which is why we have to start and then kill and remove the container. If 404 then it's not an issue to us as the container doesn't exist.

Learned

This was a type of project neither of us have done before. We both have done HTML, CSS, and Javascript but nothing connecting the two sides of server and client with a full dive into full stack development. It was a new experience for both of us, we knew what a POST and GET request were and have done something similar in the past yet, nothing as full scale as this project has been. That's one of the main reasons why we decided to go with Node.js, it's simple and easy to not only implement but also run and test.

Incorporating a front end framework so late into our development cycle also brought many challenges that we had to overcome. We learned very quickly how drastic some of the changes that come with changing an entire front end like this. One huge learning curve for us was the switch from javascript to typescript, as that is what Angular uses. Its stricter typing system and other subtle differences caused many issues. If we were to do things differently we definitely would have implemented the front end framework earlier in development. Not only did the switch cost us a lot of time due to the conversion of javascript to typescript, but it also changed the way we had to do the UI.

One of the biggest learning curves we encountered was managing Docker containers and automating many of their processes, like container creation and deletion. The most time-consuming challenge was dynamically creating containers as a user progresses through flags, while ensuring each container was attached to the correct image. Through many failed attempts and hours of research, we were able to perfect the life cycles of the containers, ensuring they were created, maintained, and deleted properly. Integrating these containers with the in-page terminal then brought its own layer of complexity that took a lot of research to fine tune.

Our Programming Strategy

We had a very basic strategy when it came to developing this project. Usually what happens is when a team is using Git in order to create a project, some team members will work towards the same files and have many merge conflicts. With just us two, we have had very little merge conflicts. One of us would work towards a feature that required some files while the other worked on a different feature that required different files. This way, merge issues are not that common as our code doesn't cross in the same file, only exception being our server file.

Doing it this way it would seem like what each of us wrote doesn't get touched by the other but that's far from accurate. After either of us implement a new feature or tweak some of the code we take the time to explain to the other what it does, why we need it, and why we did it that way so we both understand exactly what is going into the project itself.

We thought by programming this way and using Git, it was really easy to get each other's code and change things when needed. We don't think any other way would have been easier on us and easier to create the project then the way that we went with. When creating new features we also passed ideas back and forth not only on how it can be implemented but also on how the feature would look and benefit the user.

Conclusion

We both enjoyed working on this project. Transitioning from a smaller software engineering project for a semester to what it is now, is something both of us did not see coming. From the setbacks we had, to the clever solutions we made, it's been a great project to work on. Originally, it was just a project for us to build for a class, but now, as it has grown, this is something that schools and universities alike can utilize. We never intended on taking this further than what it was last semester but as we both worked on it more and more, and with the backing of Dr. Marquardson and Dr. Kowalczyk, we realized that this has some real potential. We both plan on continuing this project as we both believe we can take this project further and even be used by faculty and students.

With this project, we gained more skills not only on the technical level, but also with group work which is a prominent point in software engineering. It gave us exposure to working with full stack development as well as having a solid project to put on our resume for future job opportunities. We also got to try out some technologies we have never worked with including: Docker, MySQL, Node.js, Angular, Typescript and others. With these new skills it just improves our position as programmers and software engineers. It was a fun project to work on and as we continue to develop more in the future, we are excited and passionate for what this project can truly become.