CS 4624: Multimedia, Hypertext, and Information Access

Final Report

April 27, 2023

Traffic Simulation Management System

Virginia Tech

Blacksburg, VA 24061

Client: Dr. Mohamed Farag

Professor: Dr. Mohamed Farag

Team: Nathan Borgese, Sara Grammer, April Monk, Jack Thomas

Table of Contents

Table of Figures

List of Tables

1. Abstract

The Integration Traffic Simulator is software used by researchers, traffic planners, and traffic engineers from all over the world. The software can be helpful to simulate important factors such as safety or environmental risks after being given input data such as road networks, speed limits in the network, number and types of cars that usually travel on the network, etc[1]. However, the software is currently only able to run one simulation at a time with no way to store metadata and no easy way to re-run old simulations. This means that using the system, especially for someone who regularly uses the simulator and might have to run hundreds or thousands of simulations, could take an exceptionally long period of time and old simulations could get misplaced. To fix this problem, our team was tasked with creating a web application. The goal of the application was to provide one convenient system capable of easily running new simulations and storing hundreds of old simulations with all their information for later reference. We successfully implemented a fully usable system that allows users to run new simulations, see previous simulations, view the outputs of any simulation, re-run old simulations, download simulations as a ZIP file with everything needed to reproduce them, use custom versions of Integration, and more. We used React for our frontend, we used Node.js for the backend, and we connected the two using an API. MongoDB is used to store simulation metadata while the filesystem stores actual simulation files. In this report, we will fully discuss the requirements, design, and implementation for the application. In addition, we have included a user manual explaining how to use the application's features and a developer manual with all information required for installation and future development.

2. Introduction

The Integration Traffic Simulation Software is a traffic simulation and optimization model that can model networks of up to 10,000 links and 500,000 vehicle departures[1]. The software creates a model that traces the movement of traffic, showing information including, but not limited to, car-following, lane-changing, and distance between cars. This software is used by researchers, transportation planners, and traffic engineers.

The Integration software includes an executable which reads from a master file. This master file gives the path to a file holding input files and another file holding output files. The master file also includes the names of which input files (.dat extension) and output files (.out extension) should be used for that run of the simulation. The input files may include data like roads, speed limits, lanes, number of cars, types of cars, start/end locations, and special conditions (crashes). Some outputs of the simulator include the times of a car stopping, safety data like crash risk, and environmental factors such as fuel consumption and emissions.

Some problems have been noted, however, with usability and reusability of the Integration software. These issues are not with the functionality of the software itself, rather they are improvements that could be made in the way the software is run. Perhaps the biggest issue with the software is the lack of metadata storage. The current state of the software has the executable look at a master file to see which files to use in its execution and then read data from input files, run the simulation, and output data to output files. However, if another run is done with, suppose, the same master file but a different executable, results will overwrite those output files. This can happen because all the information for a particular run is not actually ever stored *together*, so it would be helpful to have it all packaged in one file showing the time of the run, which master file was used, and which version of the executable was used, as well as including the input and output files used and written to by that run of the simulation. Because of this lack of storage, it is also not simple to re-run a simulation with the exact same input, output, and executable files at some time in the future, as there is no record of previous runs since these are not all being stored together with a timestamp. It would be difficult to track that run in the past.

To solve some of these issues and add some new and improved features, our team was tasked with building a web application for the Integration traffic simulator. The web app will be capable of storing all metadata and other related information for a run of the simulation, including time, input and output files, master file, and the executable. Users of the web app will be able to upload the necessary files and optionally also upload an executable. The web application will show a list of previous runs, with the user having the ability to select a previous run to be re-run.

3. Requirements

The web application will need to handle a variety of tasks, including managing file uploads, running simulations, listing simulations, storing metadata, and handling a user and admin login system.

3.1 Handle File Uploads

The integration software runs based on several files: executable file, master file, input files, and output files. Thus, in order for a user to run the software from the web application, they must be able to input at least some of these files. A user of the software should be able to select input files to upload, including a master file. Then, the user should be able to either choose to upload their own executable file, select an executable from a list, or run without making any selections and use the default executable (something that can only be uploaded in admin mode) from a list.

3.2 Run Simulations

The web application's primary purpose is to run simulations. It will need to be capable of running the standard single simulation. The simulation will be run using the master and input files uploaded by the user, in conjunction with either a user-uploaded executable, a user-selected executable from a list, or the default executable if nothing is specified. As the simulation is running, output files as listed in the master file will be created and written to, as well as any errors being written to a file runerr.out. These output and error files will be available to be selected and displayed on the web app in the Run Info page as the simulation completes its execution.

3.3 Store Metadata

After a simulation is complete, the web application will store all data associated with that simulation. This data will include the user-uploaded input files and master file, the executable file used for the simulation, the output files created and written to according to the master file, the error output file runerr.out, and anything else Integration created. The data needs to be stored in a database, all together (perhaps in a zip file) with a timestamp showing the date and time at which the simulation was run. Additionally, the database entry needs to have a field to denote the ID of the user that ran the simulation. If no user was logged in, this field should be null. If a user was logged in, the ID associated with that user will be stored in the database with all of the run information.

3.4 List Simulations

Once simulations have been run and their metadata is stored in the database, the web application needs to display this for users to be able to access. To do so, the app should show a list of simulations that are running or have been previously run. Each run will show its run status (Done, Error, Running) as well as the name given to the run when it was created and the date and time that the run was completed. From this list, the user should be able to click a button to re-run that simulation or click on a run to select it for more information. Selecting a run for more information needs to display the names of input files used, the name of the executable file that was uploaded or selected, and the name of the master file. Also visible on this page should be all of the output files and the error files with any error messages, with the ability to select any of these output files and view their contents. Additionally, from here the user should be able to download a ZIP of all files associated with this run (input, master, output, error), with an option to include or exclude the executable file used, as this file would make the ZIP much larger.

3.5 Login System

A login system is also needed, with different capabilities for users logged in to the system in administrator mode versus in user mode. One similarity will be that whether a user is logged out, logged in to user mode, or logged in to admin mode, the simulation will always be run as detailed above: the user must upload input files and a master file and either upload their own executable file, use one from a list, or select nothing and use the default executable.

3.5.1 Account Creation and Login

Firstly, the application needs to support the creation of accounts with some form of hashing for passwords for account safety. Fields like first name, last name, email, and password should be requested for creating an account and the user should be informed of any fields accidentally left blank. To create a user account, any email can be entered as the email address, but the application should also have some way of allowing for admin account creation and acknowledging the difference between the two types of accounts upon creation (perhaps by requiring some specific name like ‘admin’ as the email address when creating an admin account). Upon creation of an account the user should be logged in and the page should refresh to reflect this. Once logged in, there needs to be a button for the user to log back out, which will also log out the user and refresh the page. Once an account has been created, the user should be able to log in via a ‘Log In’ button as well.

3.5.2 Unauthenticated User

If a user is *not* logged in to their account on the system, this user’s homepage should show the list of all runs which have been created by any user who was not logged in. The user will be able to start new runs and re-run old simulations as usual, but only re-run those simulations which were started by a user who was not logged in.

3.5.3 User Mode

When logged in to user mode, the user’s homepage should only show the runs that they created themselves when logged in. The user will be able to start new runs and also re-run an old simulation that they created while logged in, but they will not have access to re-run any simulations created by another user who was logged in. They should also not have access to runs created by any user who was not logged in; they would have to log out of their account to see these runs.

3.5.4 Administrator Mode

If a user logs in to admin mode, again their homepage should show only the runs they created while logged in to admin mode. Additionally, admin mode should be the only way that a user can upload a new default executable to replace the old one stored for the web application. If code for the executable has been refined/updated, an administrator may want to set it so that all future runs with the default executable will now use the new one. In this case, the user will only upload this single executable file to replace the default. While in admin mode, the user should also be able to start new runs and also re-run an old simulation that they created while in admin mode, but they will not have access to re-run any simulations created by another user who was logged in or by any user who was not logged in.

4. User Manual

Our software supports the following features:

- Upload input files
- Upload executable
- Start new simulation
- View run status and output files
- Download run as ZIP
- List old simulations
- Re-run old simulations
- Reuse old executables
- Log-in as admin
- Log-in as regular user
- Create regular user accounts
- Set default executable (only available to admin)

4.1 Banner

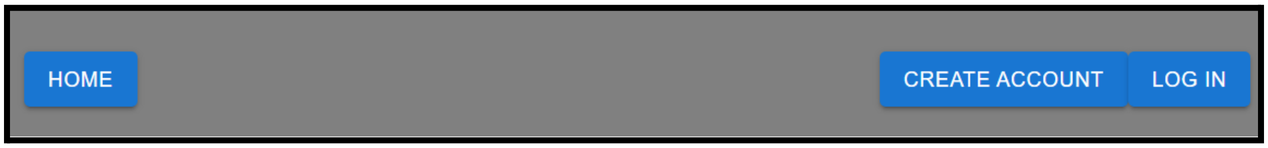


Figure 1 - Banner while not logged in

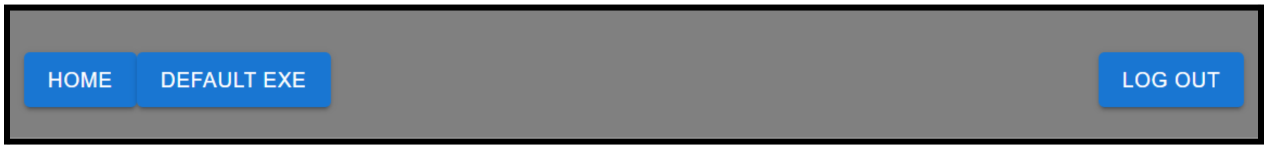


Figure 2 - Banner while logged in as admin

There is a banner that persists across all pages. This banner includes buttons to return you to the home page, and a few that are account specific. If you are not logged in then there will be two buttons for 'Create Account' and 'Log In,' shown in Figure 1. If you are logged in then there will be a button to 'Log Out' and if you are logged in as administrator then there will be an additional button to 'Set Default Exe,' shown in Figure 2. Logging in will refresh the page you are on in order to update any info that may be contingent on your account.

4.2 Dashboard

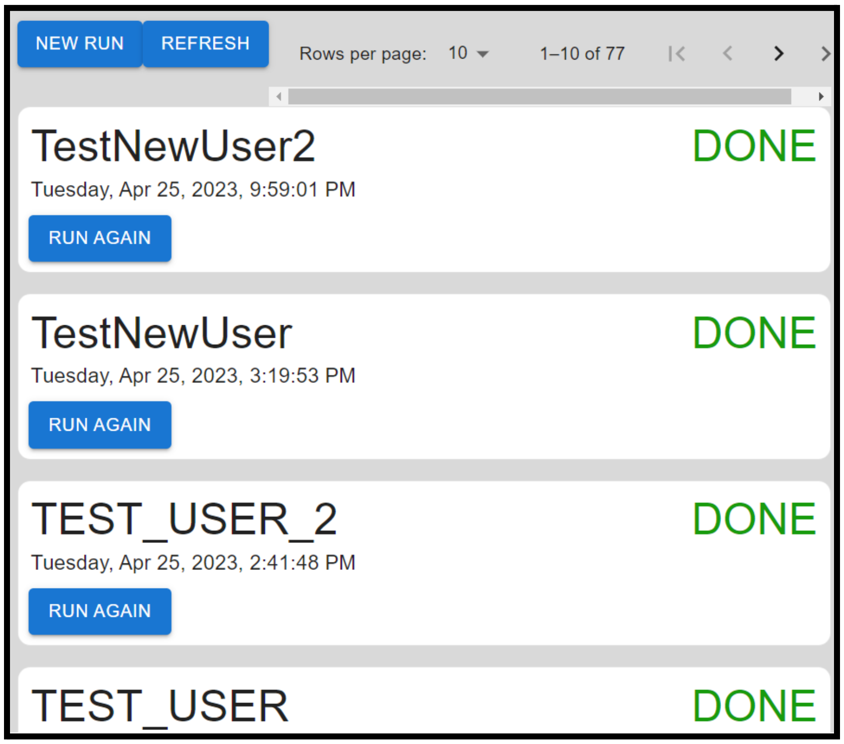
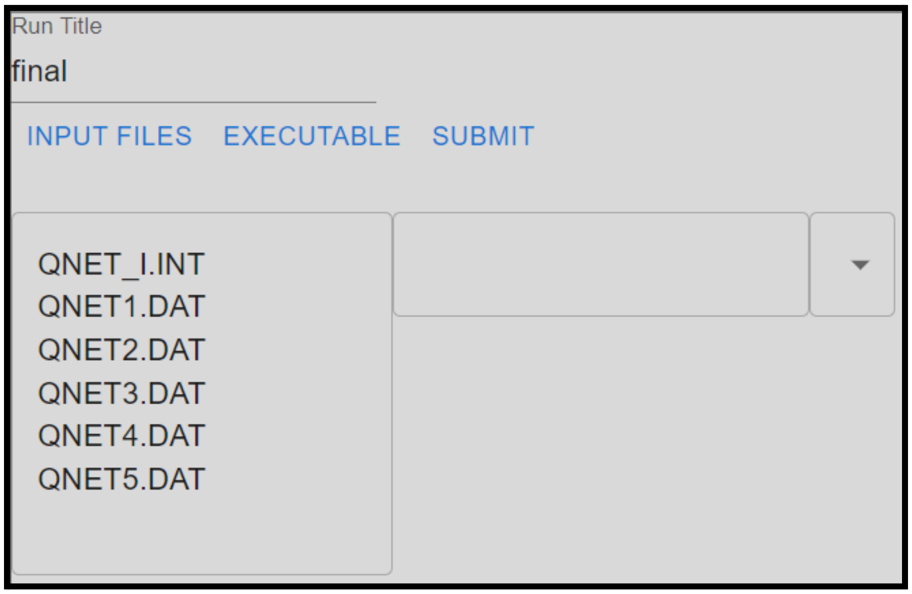


Figure 3 - Dashboard showing several previous runs

Upon loading the home page, you will see a list of previous runs as well as buttons to create a new run, refresh the list of runs, and navigate through the pages of runs, as shown in Figure 3. This list of runs will only show you runs from the account that you are logged in to. This includes showing runs created from users not logged in when you are not logged in.

Each entry for a previous run will display the run title given by the user, the time that it was run at, the status of the run (running/finished), and a button to recreate the run easily. The 'Run Again' button will take you to the New Run page with all fields filled in exactly as they were to create that run.

4.3 New Run


Run Title

final

QNET_I.INT
QNET1.DAT
QNET2.DAT
QNET3.DAT
QNET4.DAT
QNET5.DAT

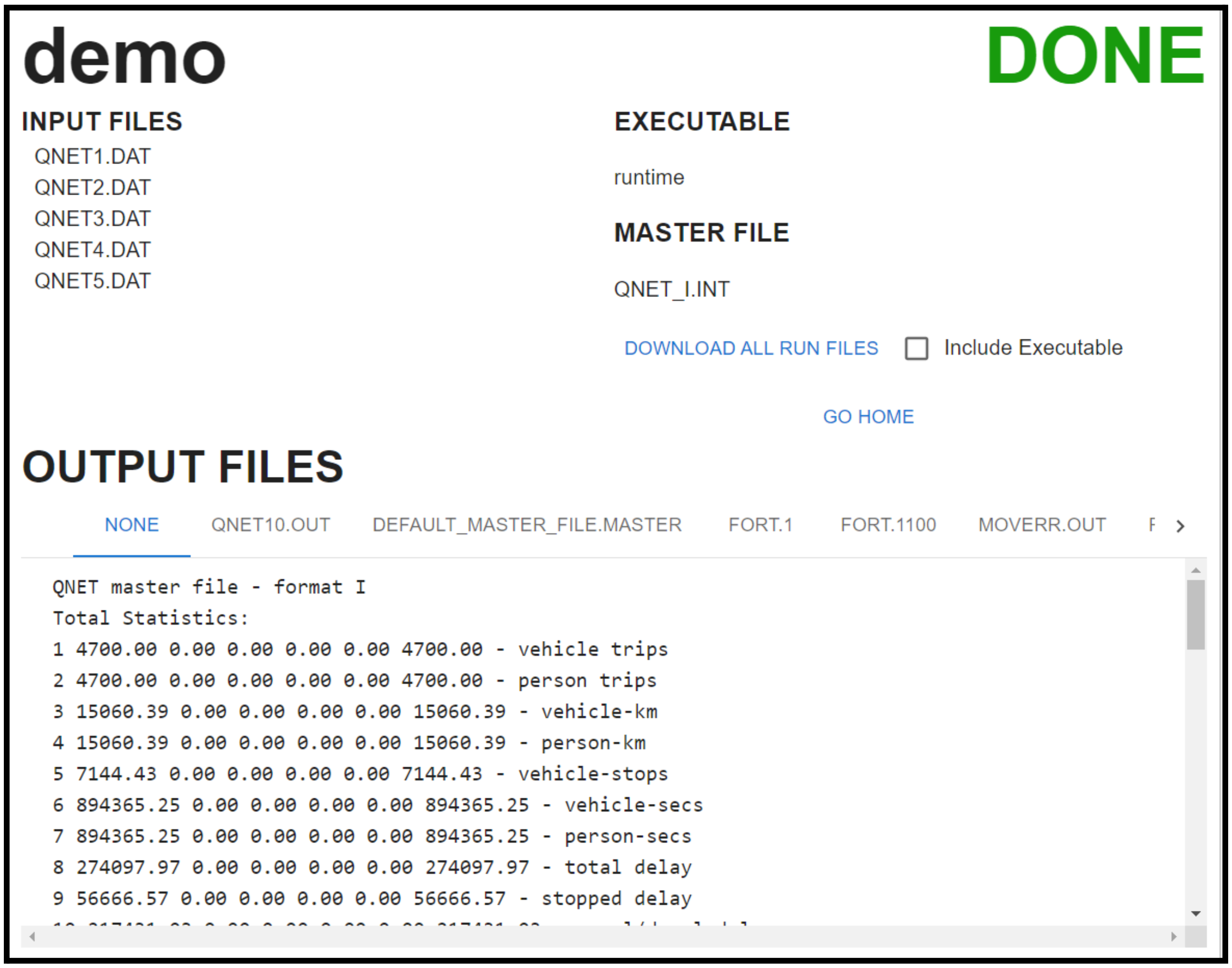
Figure 4 - New run page with input files selected and a name of 'final'

This system, shown in Figure 4, allows you to set the parameters for your new run. First, clicking the "input files" button allows you to upload the input files. You must select the master file and all the additional input files it requires. You have to select them all at once using the file selector's multi select feature. Next, you can type a name for your run in the text box at the top. Finally, you decide which executable to use. You can use the dropdown box to select from all previously uploaded executables, you can upload your own executable using the "executable" button, or you can do nothing and the default executable set by the admin will be used if it's available.

Executables must be uploaded as a ZIP file. The ZIP file must contain exactly one subdirectory at its root and nothing else at the root. The actual executable should be inside of that subdirectory, alongside any dependencies. The executable must have the `.exe` extension, even if the software is running on Linux. If there is exactly one `.exe` file, it will be used. If there are multiple `.exe` files, one of them must be called `intgrats.exe` and that one will be used.

In addition, if the server is running on Linux, the `.exe` file inside the ZIP must have execute permissions. This requires that the uploaded ZIP file was created using software that stores Linux file permissions in the ZIP. You can add execute permission to a file before zipping it by using `chmod +x filename_here`.

4.4 Run Info

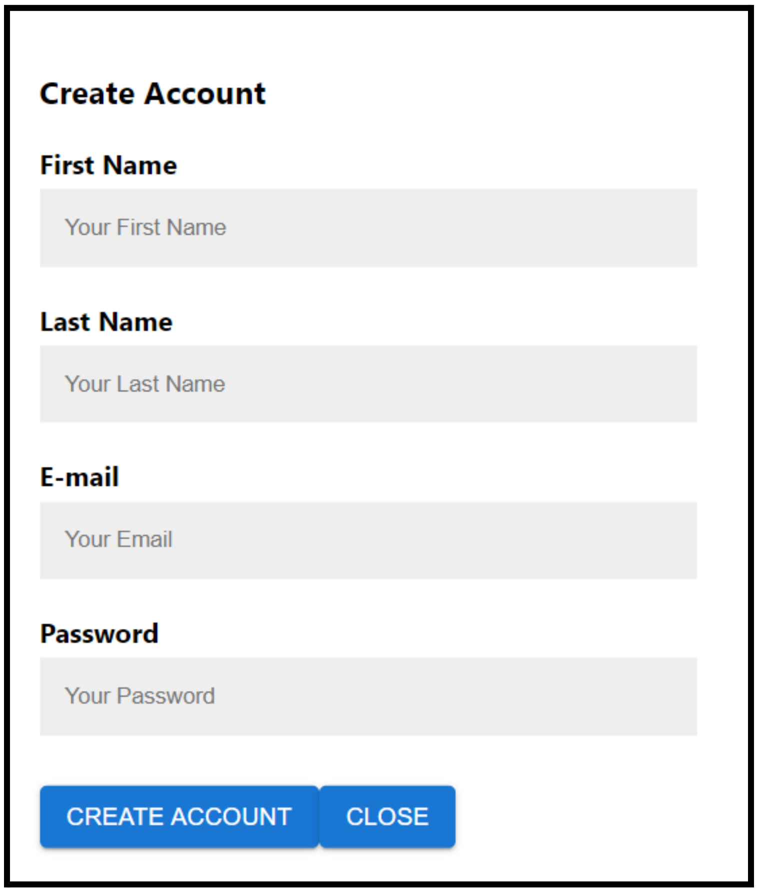


```
QNET master file - format I
Total Statistics:
1 4700.00 0.00 0.00 0.00 0.00 4700.00 - vehicle trips
2 4700.00 0.00 0.00 0.00 0.00 4700.00 - person trips
3 15060.39 0.00 0.00 0.00 0.00 15060.39 - vehicle-km
4 15060.39 0.00 0.00 0.00 0.00 15060.39 - person-km
5 7144.43 0.00 0.00 0.00 0.00 7144.43 - vehicle-stops
6 894365.25 0.00 0.00 0.00 0.00 894365.25 - vehicle-secs
7 894365.25 0.00 0.00 0.00 0.00 894365.25 - person-secs
8 274097.97 0.00 0.00 0.00 0.00 274097.97 - total delay
9 56666.57 0.00 0.00 0.00 0.00 56666.57 - stopped delay
```

Figure 5 - Run info page for run named 'demo'

After starting a run or selecting a previous run from the dashboard you will see the Run Info page, shown in Figure 5. This page shows the name of the run, the run status, the names of all input files, the executable, and the output files in full. It also has a button to download everything used in the run as a ZIP file. The executable can be excluded from the ZIP file as it increases the file size dramatically, but if it's included then a batch script to recreate the run on your local machine will be added to the ZIP file too. If the program is still running, as indicated on the page, then any output files shown may be incomplete. Once the program is done running the page will refresh automatically, displaying the complete output files and changing the run status.

4.5 Account Functions


Create Account

First Name

Your First Name

Last Name

Your Last Name

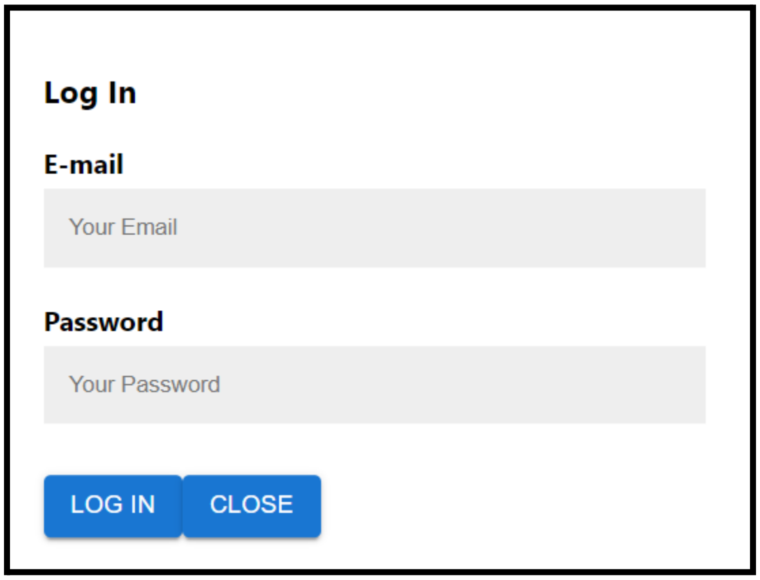
E-mail

Your Email

Password

Your Password

Figure 6 - Pop-up that appears upon clicking 'Create Account'


Log In

E-mail

Your Email

Password

Your Password

Figure 7 - Pop-up that appears upon clicking 'Log In'

To create an account, first click on the “Create Account” button in the banner. Fill out the fields shown in Figure 6 with your first and last name, your email address, and a password and then hit “Create Account.” The page will refresh and you will be logged in to your new account

To log into an account that you previously created just hit “Log In” and input the email address and password that you entered as shown in Figure 7. Again, the page will refresh and you will be logged in.

When you are logged into an account you will see some differences, notably in the home page and in the banner. On the home page you will only see runs that you created (and no one else can see those runs), and in the banner the “Create Account” and “Log In” buttons will be replaced by a “Log Out” button.

4.6 Admin Account Functions

To create the admin account, you need to create an account that has the email ‘admin’ with any first name, last name, and password associated with it. Obviously since this is the only admin account, whoever is running the instance of the management system will likely create this account as their first action.

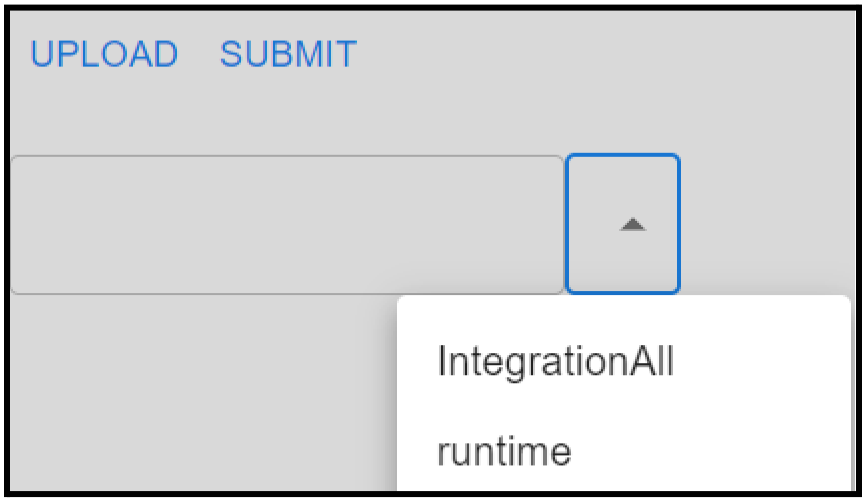


Figure 8 - Default EXE page with the dropdown showing previous executables ‘IntegrationAll’ and ‘runtime’.

The primary change that an admin account can make is to the default executable. When someone creates a run without selecting any executable it will fall back to the default executable that the admin has set. To set this, you need to click on the “Default EXE” button in the banner. From there, similar to the ‘new run’ page, you can either upload a new executable or select from a previous one, as shown in Figure 8. See the section about starting a new run for the exact requirements for the uploaded executable.

5. Design

When we began designing this web application, the first decision we needed to make was what the best stack would be for the purposes of this project. We considered a few options but ultimately decided to go with a MERN (Mongo, Express.js, React, Node.js) stack. We chose this stack because it is a common and increasingly popular stack, which meant that there would be a lot of documentation available for it. One team member already had some experience with MERN, but the amount of available documentation was appealing because it meant that it would be easier for other members to learn than some of the other stacks we considered.

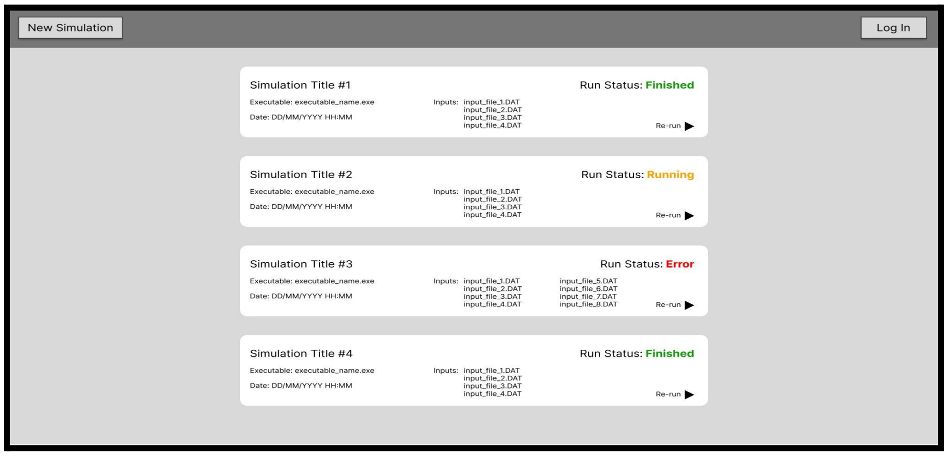


Figure 9 - Dashboard wireframe

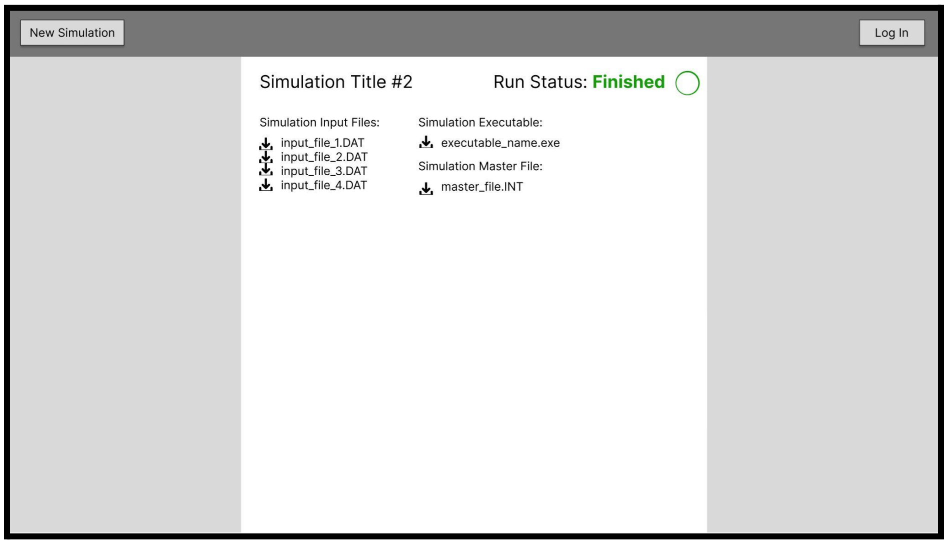


Figure 10 - Run info wireframe

The next component of the design that we wanted to discuss was the design of the frontend. We needed to make sure we understood what the frontend would look like and make sure that what we came up with was what the client wanted. We ultimately ended up creating wireframes so that we could get a clear picture of how the application would look. This helped us to understand what we would need to implement in the backend of the application to be able to create a working implementation of the design. The original design of the home page can be seen in

Figure 9 and the original design of the page for run information in Figure 10. Ultimately, the application ended up being similar to the original projection in the wireframes as shown by the figures in the previous section, which details the user manual. However, certain design aspects were adjusted during the implementation of the application to fit with the requirements and requests of our client.

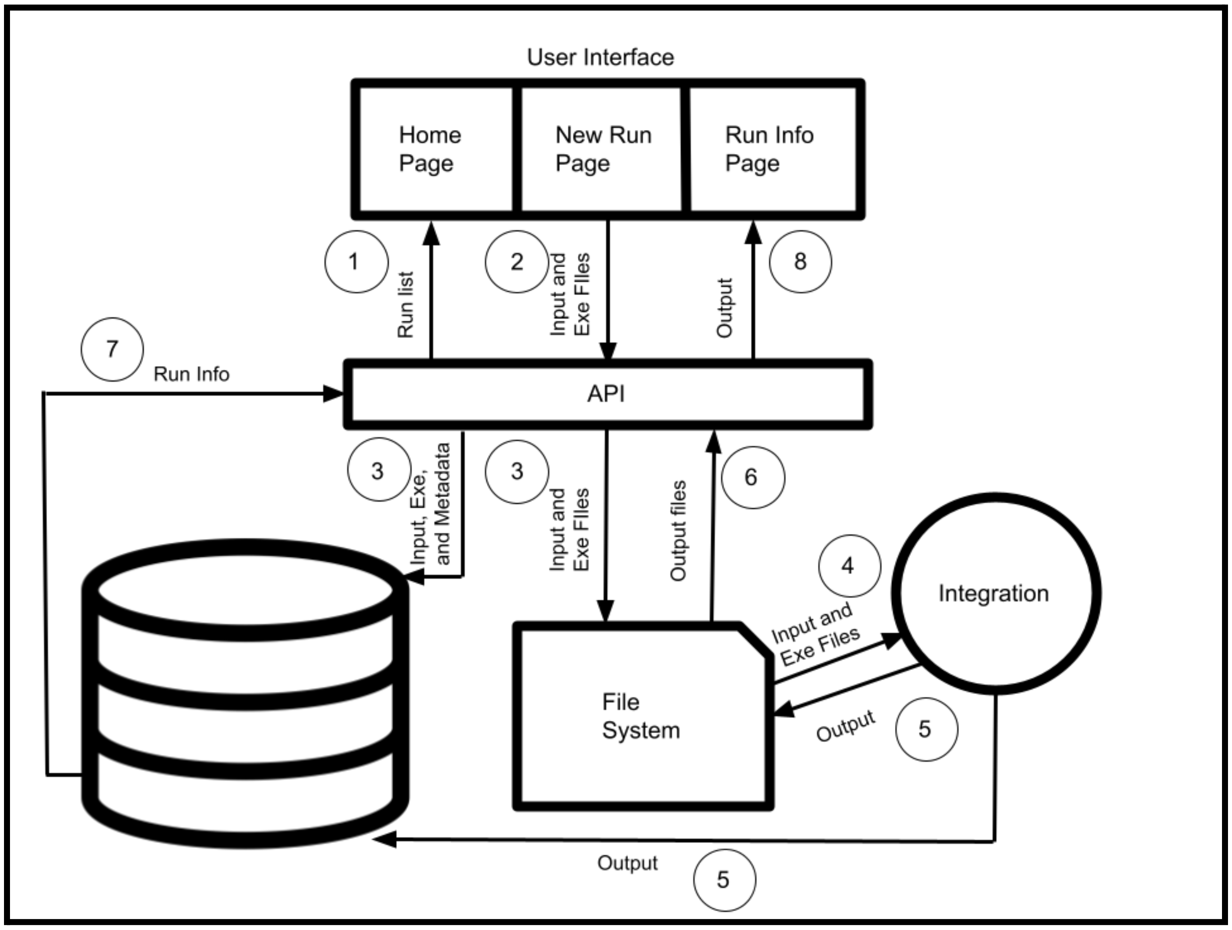


Figure 11 - System architecture

The initial design of the front end gave us a good idea of how to design the architecture for the application. As shown in Figure 11, we originally decided that we would need to have three web pages: the home page with login abilities and a list of all previous runs, a page where a user can upload new input files and executables and run the traffic simulator, and a page which contains the information for an individual run. Later, we also decided to add a page that is only available to the administrative user, which allows them to update the default executable for all application users. All of the information that goes into or out of the web pages goes through the backend, which utilizes an API that we built. The web pages give information like the inputs and executables which go through the API and into our database and our file system. That information goes from the file system to the Integration software to be run. The results and output information for that run then go back into the file system and database, go back through the API, and are ultimately displayed on the web pages.

6. Backend Implementation

The backend handles storing all persistent data and running Integration. This is the list of components used by backend and the purpose of each one:

- MongoDB - This is the database used to store persistent information.
- Filesystem - Persistent files are stored on the filesystem instead of in the database.
- Server software
  - Node.js - The backend server is programmed in JavaScript. Node.js is a JavaScript implementation that can be used to write real programs instead of just code for browsers.
  - Express.js - This is a framework for Node.js providing additional functions that make server programming easier.
  - MongoDB Node Driver - We use the official Node.js MongoDB library to interact with the database.
  - Formidable - This is a JavaScript library that makes handling user file uploads easier.
  - Bcrypt - This is a library for secure hashing. We use this for the login system.
  - Archiver - This is a library for creating ZIP files. We use this to enable ZIP file downloads of runs.
  - Decompress - This is a JavaScript library for extracting ZIP files. We use this for processing executable uploads.

6.1 How to Run Integration

The fundamental goal of the backend is to run Integration. We'll first explain the relevant details of how Integration is used so that the backend explanation makes more sense.

Integration can be used from its GUI or from the command line. For this project, we only care about the command line usage. The user will not be able to see any GUI. The command line usage of Integration is as follows:

```
intgrats.exe <path to master file>
```

The "master file" describes the properties of the simulation, which additional input files it requires, which additional output files it should make, and where the inputs and outputs should be. Throughout the simulation, Integration creates certain output files in both the output directory and the current working directory.
Our goal is to get input files, run Integration, gather the outputs, and save everything related to the simulation for later use.

6.1.1 Master File

Figure 12 shows an example master file. These are the important parts for us:

- Master File Routing Format - This is the file format version for the master file. There are three different versions: Format I, Format II, and Format III. Each format has the information on slightly different lines, so we need to know the format in order to interpret the file. The Master File Routing Format is the fourth number on the second line. In this example, it's "1", meaning Format I.

- Input and output paths - The master file specifies the path where the additional input files can be found and the path where the output files should go. The paths are relative to the location of the master file on the filesystem. In this example, the third and fourth lines are the input and output paths respectively. For other Master File Routing Formats, they're on different lines.
- Additional input files - The lines in the example ending with `.dat` are the additional input files that this simulation requires. They can be found inside of the specified input files directory.
- Additional output files - Same as the additional input files. The lines in this example ending with `.out` are output filenames.

```
INET Master File
5400 -1800  -900     1     0
inet\
inet\output\
INET1.dat
INET2.dat
INET3.dat
INET4.dat
INET5_2.dat
none
none
none
none
INET10.out
INET11.out
INET12.out
INET13.out
none
INET15.out
INET16.out
none
none
none
lane_stp.dat
det_loca.dat
scn_view.dat
```

Figure 12 - Example master file[2]

6.2 General Flow

The frontend and the backend are separate servers. The frontend JavaScript code communicates with the backend by directly sending web requests to special API urls.

Here are the most important APIs. This is also an overview of the data flow.

- Use `/api/uplad_input` to receive input files uploaded by the user.
- Use `/api/doit` to run the simulator with a specific set of input files.
- Use `/api/run_info` to get information about a run.

- Use `/api/get_output_file` to retrieve a specific output file so it can be shown to the user.

6.3 Uploading Input

The `/api/upload_input` backend endpoint accepts file uploads. It must receive a master file and all the additional input files the simulation requires. Together, this is a "set of input files." These are the steps taken when a set of input files is received:

1. Verification - The backend does some basic validity checks.
2. input_uuid generation - Whenever a set of input files is uploaded, a unique ID is generated for it. This input_uuid becomes the permanent identifier for that particular set of input files.
3. Master file modification - The server modifies the paths contained inside the master file that represent the filesystem locations for the input files and output files. Because our management system abstracts away the concept of the input and output directory, the user's original choice is irrelevant and we can choose these paths freely. When we get to the section about running the simulator, we'll detail what the new input/output paths are. Modifying the paths requires two pieces of information. First, we need to know which of the files the user uploaded is actually the master file. We currently just rely on the file extension being `.INT`, which is the recommended extension[2]. Second, we need to know the Master File Routing Format to know which line of the file the input and output paths are. We parse it as previously described in the section about the master file format.
4. Input file moving - All sets of input files are stored inside of `inputstashdir`. Inside of `inputstashdir`, there is one folder for each set of input files. The input_uuid for that set of input files is used as the name of the folder. Figure 13 shows an example.
5. Database entry creation - In order to keep track of them, one database entry is created for each set of input files. This is the format of the database entries for input:

```
{
      _id: <input_uuid>,
      masterfname: <filename of master file>,
      unixtime: <Unix time they were uploaded>
}
```

6. Success response - If none of the previous steps failed, the backend responds to the frontend with the generated input_uuid. The frontend can use this with other APIs.

```
./inputstashdir/
 ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac
      ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET1.DAT
      ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET2.DAT
      ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET3.DAT
      ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET4.DAT
      ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET5.DAT
```

```
     ./inputstashdir/868fa530-54e9-4e0d-81bf-c958a8e288ac/QNET_I.INT
./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET_II.INT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/DET_LOCA.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET1.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET2.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET3.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET4.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET5_1.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/INET5_2.DAT
     ./inputstashdir/9162920b-5f09-41f1-8872-9560e8b93616/LANE_STP.DAT
```

Figure 13 - Example `inputstashdir`. It contains two sets of input files.

6.4 Running Integration on the Backend

The `/api/doit` endpoint takes an input_uuid as input, representing a set of input files, and runs Integration with that input. These are the steps required to run the simulator with a set of input files:

1. File structure creation - Similar to as described in the section about input uploading, a unique output_uuid is generated for this specific run of Integration. An output folder is made too with that output_uuid as its name. This is the folder's structure:
    - `outputstashdir/<output_uuid>/output` - This directory contains the regular output files generated by Integration.
    - `outputstashdir/<output_uuid>/cwddump` - This is the "current working directory" when running Integration. Besides the regular output files that Integration creates, it also creates additional output files in its current working directory. This includes the important `runerr.out` file containing errors. These additional files will be stored here.
    - `outputstashdir/<output_uuid>/tempinput` - Before we run Integration, we actually need to copy all the input files into a temporary folder.
      Remember, the path that the output files should go is hard-coded into the master file. As a result, if we had two instances of Integration running on the same set of input files, their output files would go in the same place and clobber each other. Because the output path is relative to the master file's path, we have no choice but to change the path of the master file. We do this by copying the master file and all associated input files.

      All uploaded master files have their input path changed to `/`. That's a relative path, meaning the input files and master file are stored in the same folder. That folder is `outputstashdir/<output_uuid>/tempinput` when running. The master file's output path is changed to `/../output/` so that the output files go inside of `outputstashdir/<output_uuid>/output`.

2. Database entry creation - Each run has its own database entry. This is a simplified overview of its format:

```
{
      name: <a name given by the user>,
      complete: <is the run complete?>
      input_uuid: <input_uuid is a reference to the input>,
      output_uuid: <output_uuid is a reference to the folder described
earlier>,
      user_id: <the id of the user that was logged in to make the run, if
no user logged in, this field is null>
}
```

3. Integration launching - Integration is started using Node.js' `execFile` function. This is an asynchronous function, so it doesn't block the backend. We pass it the correct path to Integration's executable, the "current working directory" to run it in, and the path to the master file which will be used to start Integration. When `execFile` detects that Integration has finished, it runs a callback function to update the database.
4. Success response - If none of the previous steps failed, the backend returns the run_uuid to the frontend. The run_uuid is a reference to the database entry, which can be used to get all information about the run.

6.5 Getting Run Info

The `/api/run_info` backend API can be used to get information about a specific run. It takes the `run_uuid` as an argument and returns everything that we'd need for a "run info" page on the frontend.
A simplified list of the information returned is shown below. Refer to the API documentation for the full response.

```
{
      overview: {
            unixtime: <time run was started>,
            run_status: "done"/"running",
            run_uuid: <run_uuid, same as the input>,
            name: <run name>,
      },
      files: {
            output: [<filenames of all files in
outputstashdir/<output_uuid>/output>],
            cwd: [<filenames of all files in
outputstashdir/<output_uuid>/cwddump>],
      }
}
```

6.6 Getting Output Files

The `/api/get_output_file` backend API can be used to get a specific output file. Its input is a `run_uuid` and an output file's filename. The output files' names can be retrieved from `/api/run_info`. If the given filename is valid, the response is just the file contents which can then be displayed on the frontend.

6.7 Other Features

It would take too long to detail the implementation for all features and it would be tedious to read. This section provides a brief overview of how some of the other features are implemented:

- Custom executables: Executables are uploaded similarly to input files. Each executable uploaded has its own executable_uuid, which is passed to `/api/doit` to tell the backend which executable to use.
    - Set default executable: We have a file on the backend that stores the executable_uuid of the default executable. The file is modified using a special `/api/set_default_exe` API.
- Download run as a ZIP file: We use the "archiver" library to create a ZIP file. The ZIP file is not saved to disk first, it is dynamically created and sent over the network as it's generated.
- Account creation/login: Account credentials are stored in the database in an uninteresting way. We *do* hash passwords, but *they are not salted*. When the user logs in, the backend returns a session token to the frontend which serves as proof.
    - Per-user data: To allow a user to have his own runs, we simply pass the login session token as a parameter to the APIs. If the backend finds a valid token passed to the API, it will find which user it corresponds to and augment its behavior to specialize the results to that user. For example, when a session token is passed, the API for listing runs will only return runs for that user.

7. React Web App Implementation

7.1 React

The web application runs on top of Node.js, a command-line JavaScript runtime environment. It was created initially using npm's 'create-react-app'[3] library which creates a base web app that can be run from the command line using 'npm start.' React uses JSX to allow developers to write HTML-like code within a JavaScript application, enabling them to write web UI code that is adaptable and closely integrated with the existing JavaScript code. This JSX code can be turned into React components, a repeatable element that can be imported from third party libraries.

7.2 MUI Components

The library that we chose to use primarily for this project is Material UI (MUI). MUI is one of the most popular react component libraries because of its adaptability, accessibility, and availability of open-source components. Using MUI will allow us to have a project that looks good without modifying many components right away, while giving us maximum flexibility both visually and functionally. For instance, in order to handle file upload, we use MUI Button wrappers that use an HTML form upload system internally. Elsewhere we use several MUI TextFields to handle text input and output.

7.3 Stateful Components

In order for React to respond to changes, it uses application State. When a value defined in the application state or that uses useState() is updated, everything on the page that is listening for that change will update. This makes it easy to handle user input on the fly and have those changes reflected in multiple components. Additionally, some components are stateful and some are stateless. This means that a stateful component will be listening for these application state changes and will react to them, whereas the stateless component will be rendered with a set of data and stay that way unless told to rerender. Generally, there will be at least one stateful component, and good practice will lead you to having several stateless components that make up the big chunks of your application, all contained within stateful components.

8. Testing

During the process of creating this application, we felt that because many of the features we were implementing had similar components and if we waited until the end to test the whole application, we would likely be fixing the same bugs in multiple places. Therefore, we decided that the best way for us to test our work was to test each feature during its implementation and as we finished it, rather than waiting until we finished the application to test all of the features at once.

When one member of the team was working on implementing a feature, they would typically test parts of that feature as they went. One way that we went about this was utilizing console.log statements in both the front and backend and running that portion of the application to see how data was moving in a POST or GET request. We would also utilize these statements to make sure that when we needed to modify a variable or verify a conditional statement, we were producing an expected value. This allowed us to ensure, as we went through implementing a feature, that data was properly moving between the front and back ends of the application. It also allowed us to locate issues early if the implementation of a feature was not behaving as we expected.

Once a member of the team believed that the feature they were working on was working properly, they would push the modified code to our git repository so that everyone could access that code. After that, during our weekly meetings, the other members of the team would look at the newly implemented code and run the application with the updates. From there, team members would test every possible scenario for a feature. For example, when testing the feature to create a user account, what would happen if you created an account but didn't fill in a field in the form, or what if you tried to make an account with an email address that is already associated with an account. We would try to test every possible scenario in a feature that could come up. This allowed us to find any bugs or unexpected behavior that the person who wrote the feature might not have thought of when implementing the feature or doing their own testing. Once the team had tested a feature, the person who implemented it would take the notes about what needed to be fixed and would implement those changes. This would start our testing process over again and we would continue, iteratively, until we could no longer find any issues.

Once we were sure we had implemented all of the features we would be able to implement within the time constraints of the project, we all went through and tested all of the features to make sure that we would not find any bugs or unexpected behaviors that we did not previously find. We utilized the frontend of the application and once again tried all of the different behaviors that could occur in a feature. In layman's terms, we essentially tried to find a way to 'break' a feature with an unexpected input. This was to make sure that we had accounted for all contingencies and that our application did not produce any outputs or have any behaviors that we did not expect.

In doing this final round of testing, because we had tested each feature individually as we went along, we did not find many issues. When we did, they were generally pretty small issues that were relatively easy fixes. Many of the issues were actually just small things that we didn't necessarily need to change but that we wanted to add to create the best possible user experience.

9. Developer Manual

9.1 Installation and Running

9.1.1 Prerequisites

- Windows or Linux

  We've tested on Windows 10 and CentOS 7. Any OS which you can install the dependencies on and run Integration should work.

  NOTE: We did not get the opportunity to try the system with a Linux version of Integration. However, we believe it will work.

- You must have three ports available. If you want the server to be accessible by others, two of the ports need to be accessible from the outside. They can't be blocked by a firewall. If you only care about running locally, you can keep the firewall.

9.1.2 Installing Software

- Node.js
  - Windows

    Download and install Node.js v18.15.0 for Windows. Here is a direct download[7]:
    https://nodejs.org/dist/v18.15.0/node-v18.15.0-x64.msi
    All the installer's default settings are fine. Don't remove any features on the "custom setup" part. Make sure to check "Tools for Native Modules" when you get to that part.
  - Linux

    Refer to the official Node.js installation instructions[4]:
    https://nodejs.org/en/download/package-manager
    If your Linux distribution isn't listed, look at the alternative binary downloads at the bottom of the page[4]:
    https://nodejs.org/en/download/package-manager#alternatives
    We've tried Node.js v19, v18, and v17. Try to get the latest version your OS supports.

  To verify Node.js is installed correctly, open a new PowerShell/Bash window and enter `node`. It should say `Welcome to Node.js`. Press ctrl+D to exit Node.js.

- MongoDB
  - Windows

    Download and install MongoDB Community Server version 6.0.5. Here is a direct download[5]:
    https://fastdl.mongodb.org/windows/mongodb-windows-x86_64-6.0.5-signed.msi
    This will install MongoDB as a service that automatically starts and runs on boot. On the "choose setup" part of the installer, select "complete" instead of "custom." You can keep the rest of the installer's options as their defaults. If you want, you can uncheck the "Install MongoDB Compass" option when you get there.

  - Linux

    Download and install MongoDB Community Server.

1. Visit the download page[5]:
   https://www.mongodb.com/try/download/community
2. Scroll to "MongoDB Community Server Download."
3. Set the version to 6.0.5.
4. Set the platform to match your distribution of Linux.
5. **Set the "package" to "tgz."**
6. Extract the downloaded file using the tar command:
   ```
   tar xf mongodb-linux-x86_64-rhel70-6.0.5.tgz
   ```
   The second argument is the filename. It will be different depending on what you downloaded exactly.

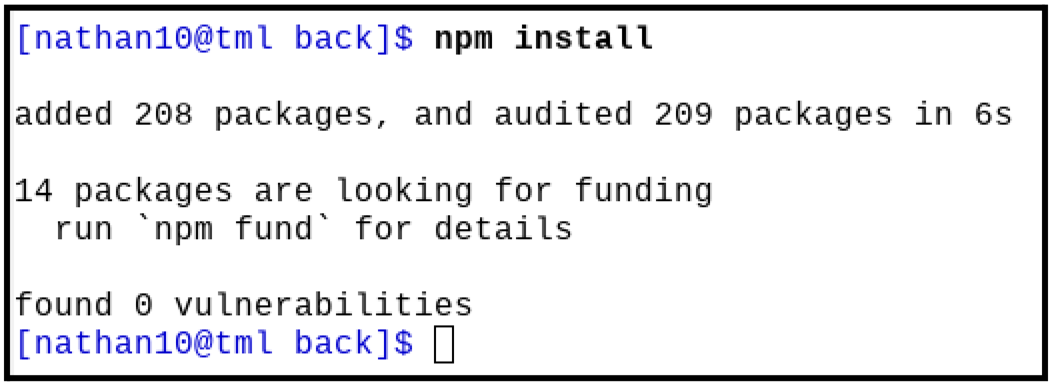
You now have the MongoDB server executables. The later section on running the server will teach you how to run it.

- Libraries

  This step will install the JavaScript libraries.

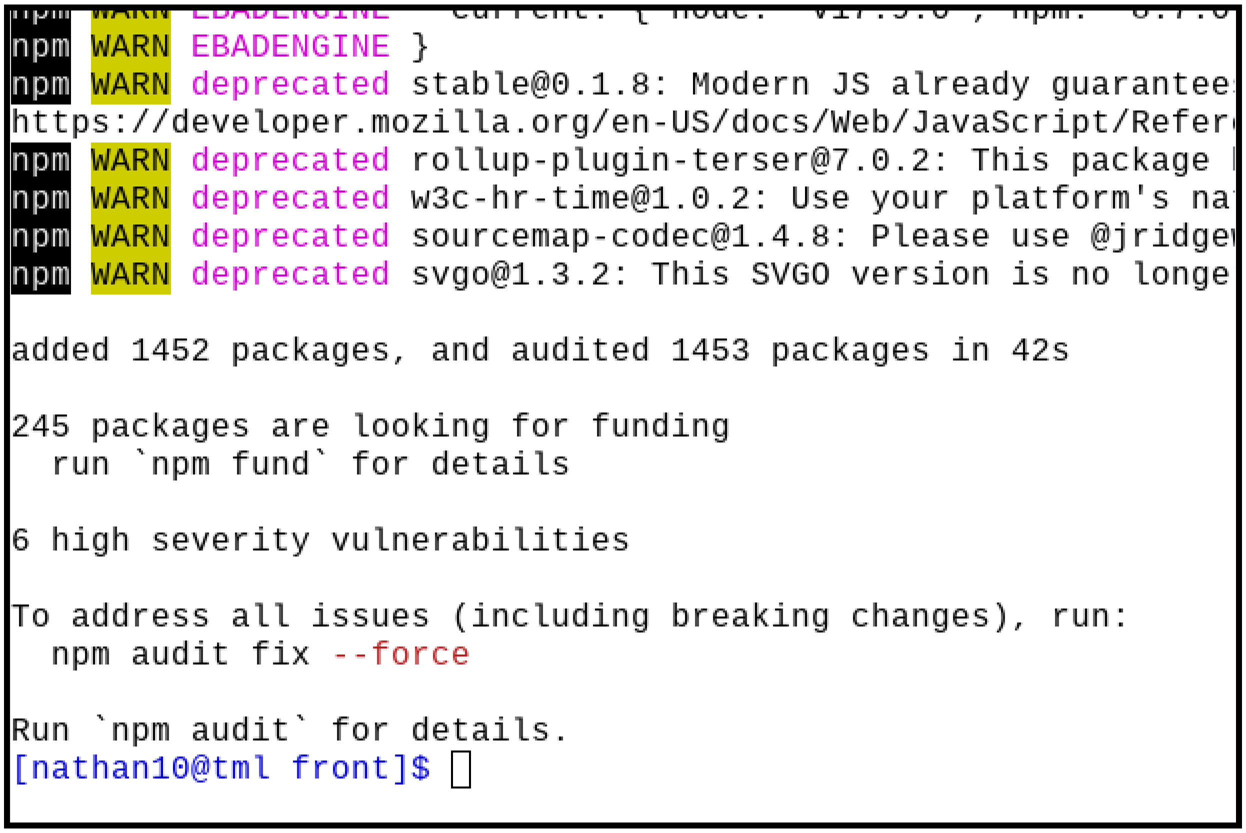
  1. Open PowerShell/Bash inside of the `traffic_simulation/back` directory and enter `npm install`.
  2. Do the same with PowerShell/Bash inside of the `traffic_simulation/front` directory. This could take a couple minutes.

  There may be depreciation, `EBADENGINE`, and other warnings. It would be good if somebody fixed them, but they're not important for this project right now and you can ignore them. Figures 14 and 15 show the expected output for `back` and `front` on Linux if the installation succeeded.



```
[nathan10@tml back]$ npm install

added 208 packages, and audited 209 packages in 6s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
[nathan10@tml back]$ 
```

Figure 14 - npm install success message for backend.



```
npm WARN EBADENGINE   current: { node: 'v17.9.0', npm: '8.7.0
npm WARN EBADENGINE }
npm WARN deprecated stable@0.1.8: Modern JS already guarantee
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Refer
npm WARN deprecated rollup-plugin-terser@7.0.2: This package
npm WARN deprecated w3c-hr-time@1.0.2: Use your platform's na
npm WARN deprecated sourcemap-codec@1.4.8: Please use @jridge
npm WARN deprecated svgo@1.3.2: This SVGO version is no longe

added 1452 packages, and audited 1453 packages in 42s

245 packages are looking for funding
  run `npm fund` for details

6 high severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
[nathan10@tml front]$ 
```

Figure 15 - npm install success message for frontend. There are warnings.

9.1.3 Configuration

This section will teach you how to configure the server so it will run properly. If you want to use the default ports, ignore the instructions for changing ports.

- Port configuration
    1. Decide on ports for the frontend, the backend, and MongoDB. **Write all of these down, they are very important.** Note that the MongoDB port doesn't need to be accessible by anybody but localhost.

       On the default installation for Windows, the port for the MongoDB service is 27017. You can change it by editing the config file located at `C:/Program Files/MongoDB/Server/6.0/bin/mongod.cfg` by default.
    2. Open the `traffic_simulation/back/.env` file in a text editor. On Linux, this file may be hidden by default because it starts with a dot.
    3. Edit the line that says `PORT=8080` to be your desired backend port.
    4. Change the port in the backend's URI for connecting to the MongoDB server. This is done by changing the `27017` part of the `ATLAS_URI=mongodb://127.0.0.1:27017` line in the same `.env` file.
- Frontend URL configuration

1. Open the `traffic_simulation/front/src/pages/Home.js` file.
2. Find the line that sets the variable containing the URL to the backend server.

   This line is the following by default:

   ```
   const backend_baseurl="http://localhost:8080";
   ```
3. Change the `8080` part of the line to the port of your backend server.
4. Change the `localhost` part of the line to the hostname of your server.

   If you only want to run locally, you can leave it as `localhost`. If you want this to be a publicly available server, set it to your domain name or IP address. If my domain is `traffic.com` and my port was `1234`, the entire line should be

   ```
   const backend_baseurl="http://traffic.com:1234";
   ```

9.1.4 Starting Servers

The backend, the frontend, and MongoDB are all separate servers. You need to start all of them.

- Starting MongoDB depends on how it was installed.

  On Windows, MongoDB was installed to run automatically as a service. You don't need to start it.

  On Linux, first find the `mongod` executable. You need to use `cd` to navigate into the folder created when you extracted a tar as part of the instructions in this report about installing MongoDB. If the root of the tar was a folder called `mongodb-linux-x86_64-rhel70-6.0.5`, then `mongod` is likely located at `mongodb-linux-x86_64-rhel70-6.0.5/bin/mongod`.

  Once you found the mongod executable, try to run it with `./mongod`. It should give a bunch of output, but not start running. Next, create a folder to hold the database. For this example, we'll do `mkdir ~/trafficdb`. This only needs to be done once.

  Finally, start MondoDB using a command like this:

  ```
  ./mongod --dbpath ~/trafficdb/ --port mongodb_port_here
  ```

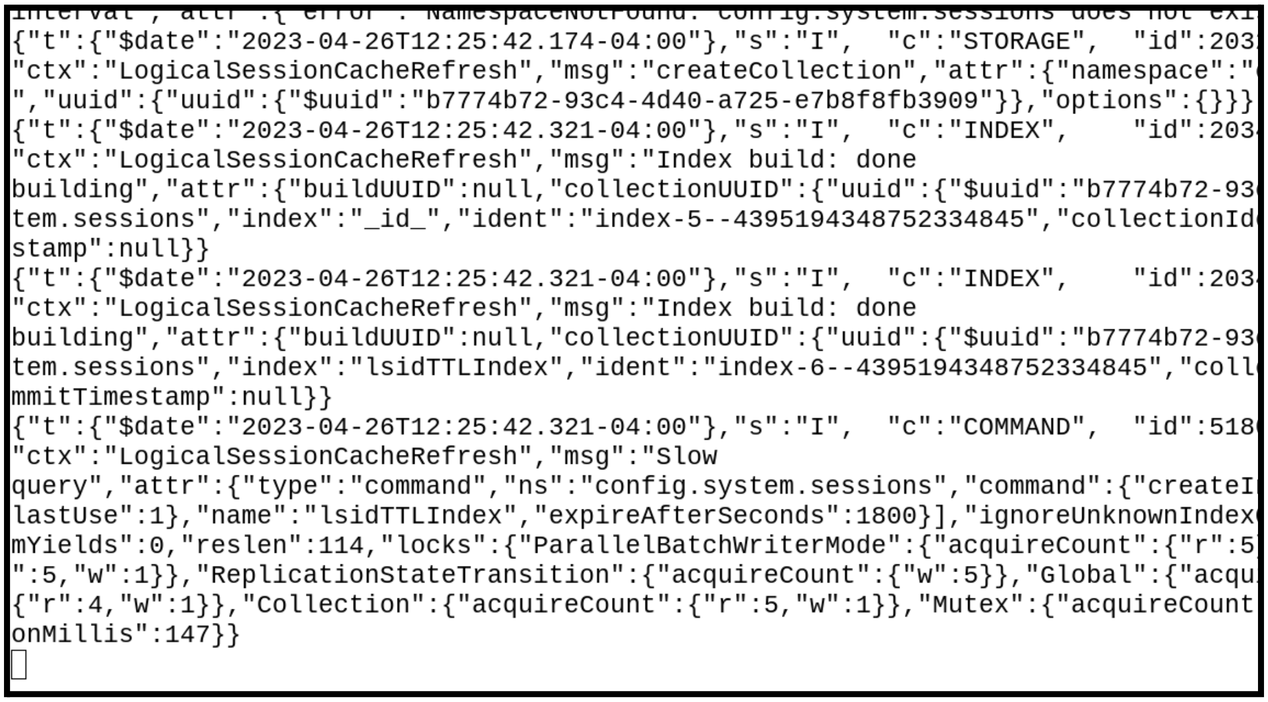
  Figure 16 shows MongoDB running. Your shell should be taken away, MongoDB runs in the foreground with this configuration.

- To start the backend server, open a PowerShell/Bash window in the `traffic_simulation/back` directory and run `node ./app.js`.

  (Windows only) If you get a popup like in Figure 17, Windows is blocking external connections to the backend server. Click "Allow access" if you intend to make this server accessible by others or "cancel" if you only want to run the backend and frontend locally. If you get an error like in Figure 18, it means the backend couldn't connect to MongoDB running locally. Make sure you started MongoDB and correctly configured the port in the URI in `.env`.
  The normal output if the server is running successfully is shown in Figure 19.

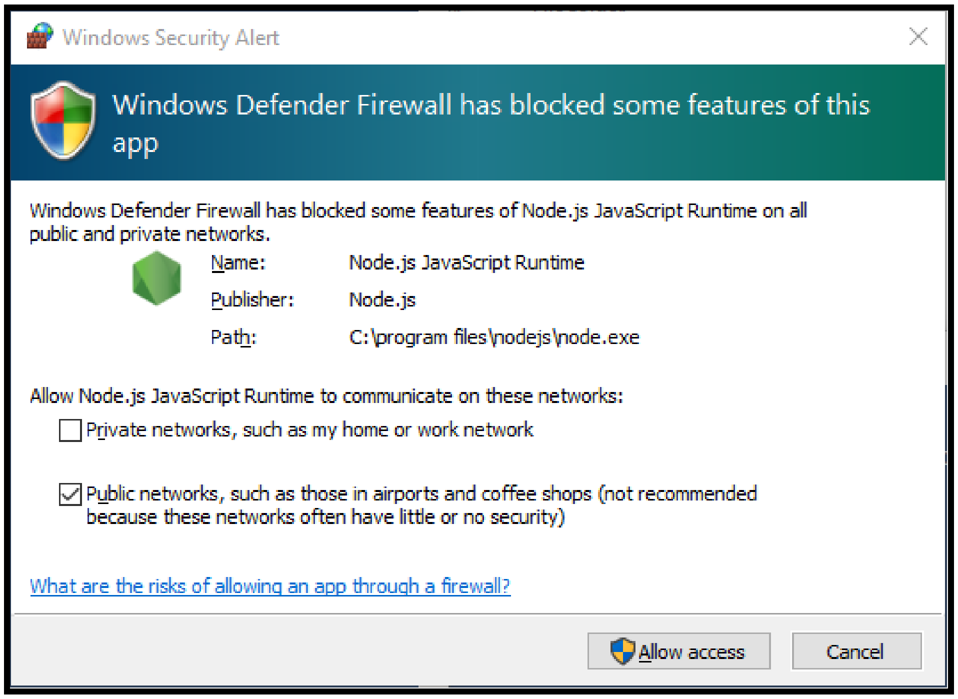
- To start the frontend server, open a PowerShell window in the `traffic_simulation/front` directory and run the following command:
  `PORT=frontend_port_here npm start`.
  The first time you start the frontend server, it will show the output of Figure 20. Sequential runs of the frontend server will show the output of Figure 21. In both cases, the server is running.

The running frontend can now be accessed by visiting `http://your_domain:frontend_port` in your browser. If you left everything as the default, then it's just `http://localhost:8080`.



```
interval","attr":{"error":"NamespaceNotFound: config.system.sessions does not exi
{"t":{"$date":"2023-04-26T12:25:42.174-04:00"},"s":"I",  "c":"STORAGE",  "id":203
"ctx":"LogicalSessionCacheRefresh","msg":"createCollection","attr":{"namespace":"
","uuid":{"uuid":{"$uuid":"b7774b72-93c4-4d40-a725-e7b8f8fb3909"}},"options":{}}}
{"t":{"$date":"2023-04-26T12:25:42.321-04:00"},"s":"I",  "c":"INDEX",    "id":203
"ctx":"LogicalSessionCacheRefresh","msg":"Index build: done
building","attr":{"buildUUID":null,"collectionUUID":{"uuid":{"$uuid":"b7774b72-93
tem.sessions","index":"_id_","ident":"index-5--4395194348752334845","collectionId
stamp":null}}
{"t":{"$date":"2023-04-26T12:25:42.321-04:00"},"s":"I",  "c":"INDEX",    "id":203
"ctx":"LogicalSessionCacheRefresh","msg":"Index build: done
building","attr":{"buildUUID":null,"collectionUUID":{"uuid":{"$uuid":"b7774b72-93
tem.sessions","index":"lsidTTLIndex","ident":"index-6--4395194348752334845","coll
mmitTimestamp":null}}
{"t":{"$date":"2023-04-26T12:25:42.321-04:00"},"s":"I",  "c":"COMMAND",  "id":518
"ctx":"LogicalSessionCacheRefresh","msg":"Slow
query","attr":{"type":"command","ns":"config.system.sessions","command":{"createI
lastUse":1},"name":"lsidTTLIndex","expireAfterSeconds":1800}],"ignoreUnknownIndex
mYields":0,"reslen":114,"locks":{"ParallelBatchWriterMode":{"acquireCount":{"r":5
":5,"w":1}},"ReplicationStateTransition":{"acquireCount":{"w":5}},"Global":{"acqu
{"r":4,"w":1}},"Collection":{"acquireCount":{"r":5,"w":1}},"Mutex":{"acquireCount
onMillis":147}}
```

Figure 16 - MongoDB running



Windows Security Alert

Windows Defender Firewall has blocked some features of this app

Windows Defender Firewall has blocked some features of Node.js JavaScript Runtime on all public and private networks.

Name: Node.js JavaScript Runtime

Publisher: Node.js

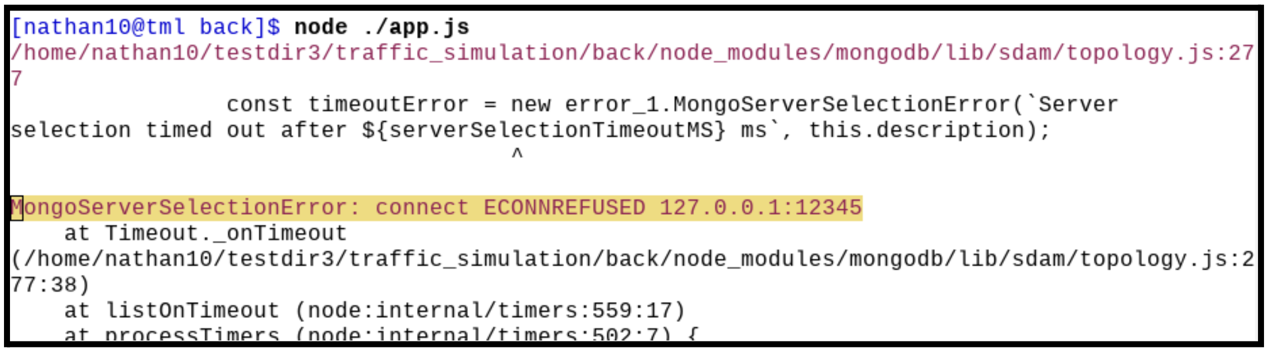
Path: C:\program files\nodejs\node.exe

Allow Node.js JavaScript Runtime to communicate on these networks:

- [ ] Private networks, such as my home or work network
- [x] Public networks, such as those in airports and coffee shops (not recommended because these networks often have little or no security)

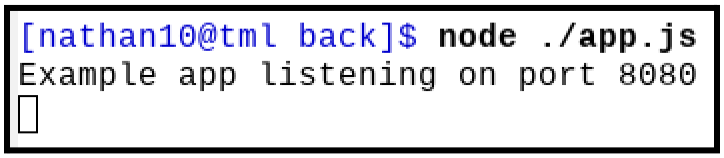
What are the risks of allowing an app through a firewall?

Figure 17 - Firewall access request



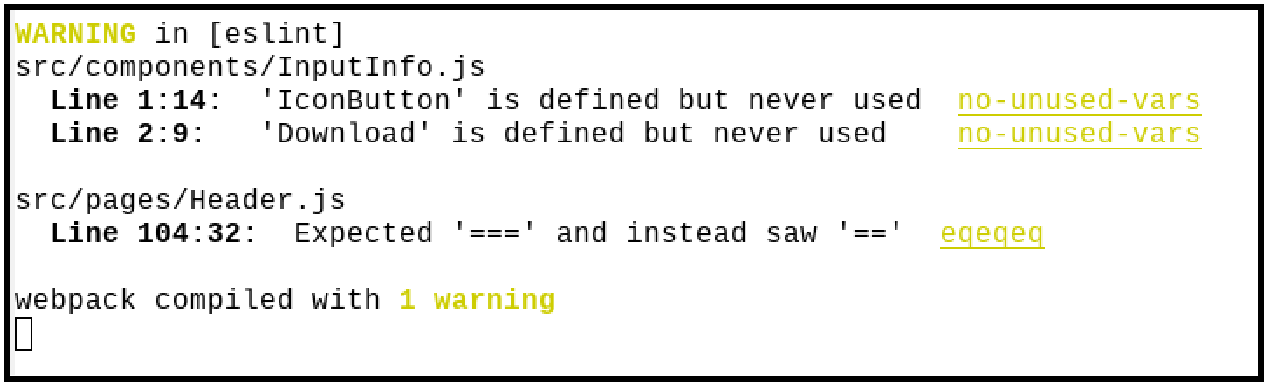
```
[nathan10@tml back]$ node ./app.js
/home/nathan10/testdir3/traffic_simulation/back/node_modules/mongodb/lib/sdam/topology.js:27
7
                const timeoutError = new error_1.MongoServerSelectionError(`Server
selection timed out after ${serverSelectionTimeoutMS} ms`, this.description);
                                      ^

MongoServerSelectionError: connect ECONNREFUSED 127.0.0.1:12345
    at Timeout._onTimeout
(/home/nathan10/testdir3/traffic_simulation/back/node_modules/mongodb/lib/sdam/topology.js:2
77:38)
    at listOnTimeout (node:internal/timers:559:17)
    at processTimers (node:internal/timers:502:7) {
```

Figure 18 - The output if the backend can't connect to MongoDB



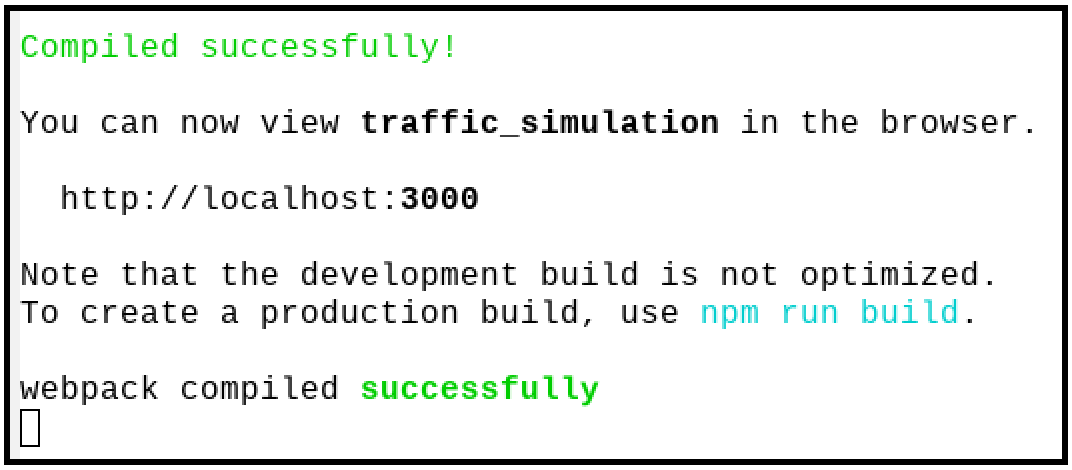
```
[nathan10@tml back]$ node ./app.js
Example app listening on port 8080
```

Figure 19 - Backend running successfully



```
WARNING in [eslint]
src/components/InputInfo.js
  Line 1:14:  'IconButton' is defined but never used  no-unused-vars
  Line 2:9:   'Download' is defined but never used    no-unused-vars

src/pages/Header.js
  Line 104:32:  Expected '===' and instead saw '=='  eqeqeq

webpack compiled with 1 warning
```

Figure 20 - npm start is stuck here, but the server is still running



```
Compiled successfully!

You can now view traffic_simulation in the browser.

  http://localhost:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Figure 21 - Frontend running successfully and it's not the first time it was run

9.1.5 Troubleshooting

Getting the ports to work properly can be hard. If the backend gives an `EADDRINUSE` error, it means the port and address combination are already in use. Pick a different port. If the backend and frontend aren't accessible from computers on the same LAN, you probably have a firewall blocking the backend or frontend's port. If the servers are accessible from LAN but not the outside world, you may need to change your router settings to open the port.

9.2 Files

Table 1 lists all important files.

File	**Purpose**
traffic_simulation/back	All backend related files go in here
traffic_simulation/back/package.json	List of JavaScript library dependencies for the

File	Purpose
	backend
traffic_simulation/back/package_lock.json	Version numbers of JavaScript library dependencies for the backend
traffic_simulation/back/app.js	All code for the backend server
traffic_simulation/back/.env	Contains the port to use for the backend and the URI for the MongoDB server
traffic_simulation/back/inputstashdir	Sets of input files are stored here
traffic_simulation/back/outputstashdir	Sets of output files for runs are stored here
traffic_simulation/back/executable_files	Extracted executable files are stored here
traffic_simulation/back/tempuploaddir	Files are temporarily stored here when uploaded before being moved
traffic_simulation/back/tempexedir	ZIP files with executables are temporarily stored here while being extracted
traffic_simulation/back/defaultexeuuid.txt	When the default executable is set, this file contains JSON with the executable_uuid
traffic_simulation/front	All frontend related files go in here
traffic_simulation/front/public	Public-facing frontend files. Consists of a fallback HTML page and related resources
traffic_simulation/front/src	React web application including all JS files and React components
traffic_simulation/front/package.json	List of JavaScript library dependencies for the frontend
traffic_simulation/back/package_lock.json	Version numbers of JavaScript library dependencies for the frontend

Table 9.2a - Important file listing

9.3 Modification

This section will help you modify the program.

9.3.1 Frontend

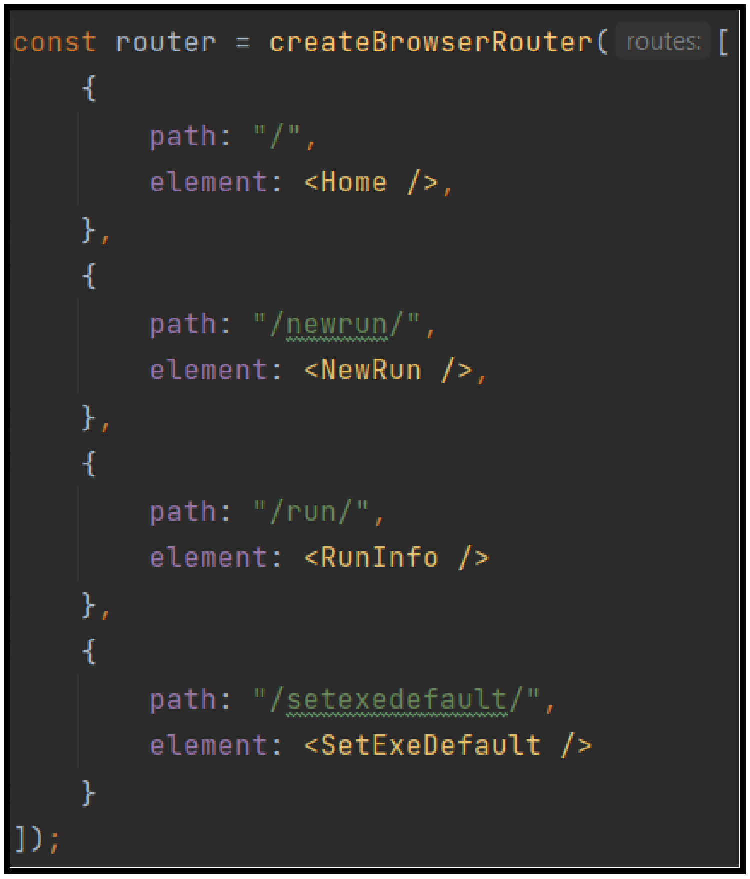
This section contains useful references for modifying or interacting with the frontend.

All frontend code is located in `traffic_simulation/front/src` except for the base HTML file stored in `traffic_simulation/front/public`. `index.js`, the file that contains the router and other low-level React implementation setup is located in the `src` folder. All pages that are used in the Router are located in `src/Pages`; components used within those pages are located in `src/Components`. Everything in the frontend is written in JavaScript using the React web framework.

9.3.1.1 Router

The React router is used to display different elements at different URLs and is stored inside of `index.js`. It is shown in Figure 22.

According to the router, when the server is running on your local machine and you navigate to `localhost:3000/` you will be shown the `Home` component. Similarly, navigating to `localhost:3000/run/` will show you the `Run` component. All main page components are stored in the `front/src/pages` folder.



```
const router = createBrowserRouter( routes: [
    {
        path: "/",
        element: <Home />,
    },
    {
        path: "/newrun/",
        element: <NewRun />,
    },
    {
        path: "/run/",
        element: <RunInfo />
    },
    {
        path: "/setexedefault/",
        element: <SetExeDefault />
    }
]);
```

Figure 22 - React router

9.3.1.2 Pages

There are currently four pages stored in the "Pages" folder. These are the .js and .css files for the Home, New Run, Run Info, and Default EXE pages. Each .js file contains at least one function that is exported which contains the majority of the code and most relevant variables and helper functions. Some other helper functions may still be exported, but the one function that exports all of the JSX for the page is the most relevant.

At the top of the function useState() is used several times in the form of `const [inputFileNames, setInputFileNames] = useState("")`. This creates a new stateful constant and a function to change the value of the constant, with a default value of an empty string. When one of those functions is called it will update that constant and push that change to the screen if that value is being used for display. Additionally, listeners can be set up using useEffect() that can run a separate function whenever that value is changed. For example, a stateful variable is used to display the value of the UUID of the simulation that can be run on the dashboard. As soon as the fetch() call returns a value, that value will be sent to the `output` variable via the `setOutput()` function. As soon as the output is updated it will be displayed on the frontend, whereas without a stateful variable the whole page would need to be refreshed or have a listener specifically for that variable set to update the TextBox.

Past the useState() calls are typically handlers for different events. Button presses, changes in text boxes, file uploads, etc. are all handled through functions defined as constants. These constants can be passed into HTML or MUI components to handle whatever those components require. No return is required from these functions, but the vast majority of them will take either values defined in the program and send them to the backend via an API call, or simply update those values.

Finally, the return value from those functions are in the form of JSX. JSX allows the return from these functions to be written in HTML-like form, to be used for display either as a component within another component, or as the main component being displayed via the React Router, taking over the base 'root' `<div>` from the HTML that is loaded onto the page. The two main types of components being used are proprietary components and MUI components. Proprietary ones are used for simplifying large chunks of code and will be expanded upon in the next subsection. MUI components are used throughout the application, including as wrappers around bare HTML components. This can be useful to capture the functionality of something like a HTML `<form>` while using a MUI component for display in order to line up with the rest of the program.

9.3.1.3 Components

There are four components stored in the "components" folder. These are the .js and .css files for the header displayed on each page, the input info and output info from the run info page, and the previous run component that contains the run info for each run on the home page.

These components follow the exact same pattern as the pages because pages are just components. The one thing that is generally different is that components may have props associated with them. While a page will receive no properties from the Router, the page may want to have some data displayed in a component that is being shown on the screen. For example, the PrevRuns.js component returns several Cards each containing a single run's data.

However PrevRuns component shouldn't make any API calls itself, instead blindly binding itself to the data that is passed in with props. Home.js will call this component in its JSX with `<PrevRuns rows={rows} />`, which then allows PrevRuns.js to get that data with `props.rows`. It can then map those rows to a series of Cards with the run name, status, time, and a button that redirects to that run's page.

9.3.2 Backend

This section contains useful references for modifying or interacting with the backend.

All the backend code, including the API implementation, is located in `traffic_simulation/back/app.js`. There isn't anything special to note about the code itself. If you know how to program in JavaScript and use Express.js, you'll be able to modify the code without any problems. If you want to look for API handlers, search for `api.get` for GET request handlers and `api.post` for POST request handlers[6]. None of the functions modify any global variables, so the code should be easy to understand.

9.3.2.1 Database Reference

MongoDB consists of multiple "databases." Each "database" consists of multiple "collections." Each "collection" can contain multiple "documents." The actual data you want to store goes inside of the documents and is stored as binary JSON (BSON).

Our program has just one database, it's called "io."
There are multiple collections. The purpose and format of their documents will be described below.

- **"runs" collection**

  There is one document in this collection for each time Integration is executed.
  Here is the format of the document:

```
{
   // ObjectId. The run_uuid of this run.
   _id:,
   // string. The name of the run. This is provided by the user, so it can
be anything.
   name:,
   // boolean. True if Integration has exited.
   complete:,
   // number. The unix timestamp the run was started at.
   unixtime:,
   // string. The input_uuid associates a set of input files with this run.
   input_uuid:,
   // string. Associates an executable with this run.
   executable_uuid:,
   // string. Associates the output directory with the run.
   output_uuid:,
   // ObjectID. If this run belongs to a user, the ID will be here. It's
null otherwise.
   output_uuid:
}
```

- **"inputs" collection**

  There is one document in this collection for each set of input files that have been uploaded.

Here is the format of the document:

```
{
   // string. The input_uuid of this set of input files.
   _id:,
   // string. The filename of the master file for this set of input files.
It's just a filename, not a full path.
   // This element is useful because it allows the backend to run
Integration on this set of input files without having to do any analysis to
find the master file first.
   masterfname:,
   // number. The unix timestamp the set of input files was uploaded at.
   unixtime:
}
```

- **"users" collection**

There is one document in this collection for each registered user.
Here is the format of the document:

```
{
   // ObjectID. The ID of the user. This is only used internally in the
backend.
   _id:,
   // string.
   first_name:,
   // string.
   last_name:,
   // string.
   email:,
   // string. A hashed copy of the user’s password
   password:,
   // boolean. True if the user is an admin.
   admin:
}
```

9.3.2.2 API Reference

This section details the different API handlers. This includes the purpose of the API, its HTTP request type (GET or POST), its expected inputs, and its expected outputs.
The implementation is **not** described here because there's too much. Refer to the section in this report about backend implementation for the main idea of the most important APIs and then refer to the source code for the concrete details.

All POST API calls that accept input are expecting it in JSON. This requires the request to have the `'Content-Type': 'application/json'` header. APIs that are GET request handlers usually don't accept any input. When they do, it comes in the query string.
Unless otherwise specified, the API returns JSON data.

Most APIs return an element called `result`. If `result` is "bad," then there was an error and the reason can be retrieved from the `errmsg` field of the response. If `result` is "good," you can proceed as normal.

- **/api/upload_input**

  Purpose: Upload a new set of input files.
  Type: POST request.

Input: The request must be an HTTP file upload request with file data, not JSON input. The master file and all the additional input files it requires must be uploaded at once.
Output:

```
{
    result:,
    // string. The input_uuid of the newly created set of input files.
    input_uuid:
}
```

- **/api/upload_executable**

  Purpose: Upload a new executable.
  Type: POST request.
  Input: The actual HTTP format of the data is the same as with `/api/upload_input`. Because Integration has thousands of files, this API expects a single ZIP file to be uploaded instead of a multi upload. The ZIP file must contain one subdirectory at its root. If the subdirectory contains exactly one executable, that executable will be used for running Integration. If it contains multiple executable files, one of them must be called intgrats.exe. That one will be used.
  Output:

```
{
    result:,
    // string. The executable_uuid of the newly created executable.
    executable_uuid:
}
```

- **/api/list_executables**

  Purpose: Return a list of all executables that are saved on the server right now.
  Type: GET request.
  Output

```
{
    result:,
    exelist: [
        // number. Timestamp the executable was uploaded at.
        unixtime:,
        // string. executable_uuid for this executable.
        executable_uuid:,
        // string. Some friendly label for this executable.
        name: elem.name,
    ],
}
```

- **/api/doit**

  Purpose: Start executing Integration with the given executable and set of input files.
  Type: POST request.
  Input:

```
{
    // string. Any arbitrary label you want.
    name:,
    // string. The input_uuid for the set of input files to run
Integration with.
```

```
    input_uuid:,
    // (optional) string. The executable_uuid for the executable to run
Integration with.
    // If not specified, the default is used if it exists.
    executable_uuid:,
        // string. The list of cookies in the user's browser.
        // If no cookies are seen present, no user is logged in, user_id
will be null
        cookies:
}
```

Output:

```
{
    result:,
    // string. The run_uuid for the newly created run of Integration.
    run_uuid:
}
```

- **/api/list_runs**

  Purpose: Return an overview of all runs. This is suitable for display on the home page.
  Type: GET request.
  Input:

```
// (optional) number. Which page of results to get.
page=
// (optional) number. How many rows per page of results.
rows=
// (optional) string. The cookies that have been or are currently
in use. If no cookies are provided, only runs created by unauthenticated
users will be returned.
cookies=
```

  Output

```
{
    result:,
    // array. Contains information about all runs.
    runs:
    [
        {
            // number. Unix timestamp of when the run was started.
            unixtime:,
            // string. "done" if run is complete or "running" if run
isn't complete.
            run_status:,
            // string.
            run_uuid:,
            // string.
            name:,
        }
    ]
}
```

- **/api/num_runs**

  Purpose: Get the total number of runs.
  Type: GET request.
  Output

```
{
    result:,
    // number. The number of runs.
    count:,
}
```

- **/api/run_info**

  Purpose: Return detailed information about a specific run.
  Type: POST request.
  Input:

```
{
    // string. The run_uuid of the run to get information about.
    run_uuid:
}
```

  Output:

```
{
    result:,
    // Object. This contains the unixtime/run_status/run_uuid/name object
that would have been returned for this run if /api/list_runs was used.
    overview:,
    // Object. Contains filenames of output files.
    files: {
        // Array of strings. Each string inside this array is the name of
an output file generated by the run.
        output:,
        // Array of strings. Same as "output" element above, but for
files Integration put in its current working directory during the run.
        cwd:,
    }
    // string. input_uuid used with this run.
    input_uuid:,
    // string. The executable_uuid used with this run.
    executable_uuid:,
    // object. This is the same as the output of /api/input_info with
this run's input_uuid.
    input_info:,
    // object. This is the same as the output of /api/executable_info
with this run's executable_info.
    executable_info:,
}
```

- **/api/get_output_file**

  Purpose: Get the content of a single output file from a specific run.
  Type: POST request.
  Input:

```
{
    // string. The run_uuid of the run that the output file is from.
    run_uuid:,
    // string. The filename of the output file to get. It must be a
filename from the "output" or "cwd" arrays from the /api/run_info API.
    filename:
}
```

  Output:

On success, this function does not respond with JSON. It responds directly with the content of the desired file. The HTTP response code is 200.
On failure, the response is a JSON object with the usual `result:'bad'` entry. The HTTP response code is 500.

- **/api/input_info**

  Purpose: Get info about a set of input files.
  Type: POST request.
  Input:

```
{
    // string. The input_uuid of the set of input files to get info
about.
    input_uuid:,
}
```

  Output:

```
{
    // string. Filename of master file.
    master_file:,
    // array. Array of filenames of all input files except the master
file.
    extra_input_files:,
    // number. Timestamp the executable was uploaded at.
    unixtime:,
}
```

- **/api/executable_info**

  Purpose: Get info about a specific executable
  Type: POST request.
  Input:

```
{
    // string. The executable_uuid of the executable to get info about.
    executable_uuid:,
}
```

  Output:

```
{
    // number. Timestamp the executable was uploaded at.
    unixtime:,
    // string. executable_uuid for this executable.
    executable_uuid:,
    // string. Some friendly label for this executable.
    name:,
}
```

- **/api/get_input_file**

  Purpose: Get the content of a single input file from a set of input files.
  Type: POST request.
  Input:

```
{
    // string. The input_uuid of the set of input files to retrieve from.
    input_uuid:,
    // string. The filename of the input file to get. It must be a
filename from the /api/input_info API.
    filename:
}
```

Output:
On success, this function does not respond with JSON. It responds directly with the content of the desired file. The HTTP response code is 200.
On failure, the response is a JSON object with the usual `result:'bad'` entry. The HTTP response code is 500.

- **/api/set_default_exe**

  Purpose: Set the default executable.
  Type: POST request.
  Input:

```
{
    // string. The executable_uuid of the executable to set as the
default.
    executable_uuid:,
    // string. The list of cookies in the user's browser.
    cookies:,
}
```

  Output:

```
{
    result:,
}
```

- **/api/demand_zip**

  Purpose: Download run as ZIP.
  Type: GET request.
  Input:

```
// string. The run_uuid of the run to download.
run_uuid=
// (optional) string. If this is "indeed" then the zip will include the
executable too. For any other value, the exe is excluded.
desire_exe=
```

  Output: On success, this API will directly return a ZIP that the browser will begin downloading. On failure, the HTTP status is 500 with the usual `result:bad` entry.

- **/api/wait_for_done**

  Purpose: Get a notification when a run finishes. A request to this API will not receive a response until the run is complete.
  Type: POST request.
  Input:

```
{
    // string. The run_uuid of the run to wait on.
    run_uuid:,
}
```

  Output:

```
{
    // string.
    // When the run completes, this API will return with result:good.
    // If the API input is wrong, the API will return with result:bad.
    // If the run status is `running,' but the run isn't actually running
in reality, the result will be result:bad.
    result:,
    // boolean.
    // This is added to the object and set to `true' if the run entry in
the database says the run is running even if it's not actually.
    // This will only ever appear when asking about runs that were active
while the backend crashed.
    // As described above, result:bad will be the status alongside
willneverbedone==true.
    // If there are no problems, this `willneverbedone' entry will be
absent.
    willneverbedone:
}
```

- **/api/createaccount**

  Purpose: Create a user or admin account.
  Type: POST request.
  Input:

```
{
    // string. The first name of the user.
    first_name:,
    // string. The last name of the user.
    last_name:,
    // string. The user's email address.
    email:,
    // string. The password associated with the user's account.
    password:,
}
```

  Output:

```
{
    result:,
}
```

- **/api/login**

  Purpose: Log a user into their account.
  Type: POST request.
  Input:

```
{
    // string. The user's email address.
    email:,
    // string. The password associated with the user's account.
    password:,
}
```

  Output:

```
{
    result:,
    // string. An object containing relevant information for an active
user session.
```

```
    session:{
        // string. The _id of the user associated with its database
entry.
        userid:,
        // string. A randomized, 15 character string to be sent to the
browser as a cookie.
        cookie:,
        // boolean. An indicator for whether or not the user is the admin
user.
        admin:,
 },
}
```

- **<u>/api/confirm_cookie</u>**

  Purpose: Confirm that the user’s browser holds a cookie for an active session. This confirms that a user is still logged in.
  Type: POST request.
  Input:

```
{
    // string. The list of cookies in the user’s browser.
    cookies:,
}
```

Output:

```
{
    result:,
}
```

- **<u>/api/logout</u>**

  Purpose: Log a user out of their account.
  Type: POST request.
  Input:

```
{
    // string. The list of cookies in the user’s browser.
    cookies:,
}
```

Output:

```
{
    result:,
}
```

10. Lessons Learned

Having worked on this project for a full semester, we have learned quite a bit about working as a team. We have found that we all have different personalities and working habits and at this point have learned to use these differences to our advantage rather than letting this negatively impact our work and progress. We found that it was an efficient method to divide the work up into current tasks and future tasks, as we had two people working on frontend and two people working on backend. This way, the frontend developers are working on two important parts of the project without accidentally overlapping their work, and the same applies for the backend.

An important thing for us to keep in mind as we worked on this project was the necessity of keeping the frontend and backend as concurrent with each other as possible. We made an effort to try and keep the two up to date with each other so that at any given point, progress was being made on both and progress could be demonstrated for our client. Early on, we would sometimes have the backend get ahead of the frontend, which worked out fine in the long-term, but would make it difficult to show our client that progress had been made between meetings as frontend progress is easier to see.

Something we made a priority was to test our code as we worked throughout the semester. Because our frontend and backend were linked from the start and stayed concurrent, anytime a change was made on either, it was reasonably simple to test if that change performed as expected and, if not, revise the code until it did. Originally, we had intended to devote the last month of the project time to testing. However, because we tested as we progressed, we did not need this extra time and were instead able to use these extra couple of weeks to meet our stretch goal of creating a functioning login system.

The reason it was particularly simple for us to test as we made progress was because the first thing we did in the project was connect the frontend and backend. This was simple to do because we chose to use the MERN stack, for linking our frontend, backend, and database, simplifying our work for this significantly. We all had at least a basic understanding of the MERN stack before the project, but we gained significant knowledge about the MERN stack, as well as more specific knowledge of React, Javascript, and linking a database like MongoDB to a web application. Though we also all were familiar with Javascript, we all learned to do new things in Javascript that we did not have prior experience with.

One major lesson we learned was the importance of efficient usage of Git. We had all used Git before, but never on a project of this caliber with this many moving parts. We gained familiarity with different Git commands, and also experienced some of the pitfalls of not making regular commits and pushes. After all, a major purpose of Git (besides collaboration) is to have a copy of all code so that code is never *exclusively* on a local machine where it can get lost in the event of computer failure.

Overall, the biggest challenge and lessons for us as a whole mostly stemmed from teamwork. It is hard, but necessary, to rely on teammates when sharing work like in this project. It can, however, be incredibly helpful to actually talk with your teammates when you have any confusion about something you are meant to be working on for the project; this will save time later if you accidentally failed to do something the way your teammates expected. If you run into an error you can't figure out, ask your teammates; chances are they have seen a similar error sometime in their work and asking for their help is hard to make yourself do but may save a significant amount of time from being wasted in searching for the bug.

11. Plans

If this project were to continue further, there are multiple features that we would want to implement. These features would increase the usability of the current application. They would allow a user to more easily utilize the application as well as expand what the application can currently do.

The first feature that we would want to implement is being able to view and edit input files that a user has already added to the application. Currently, if a user would like to modify the input files from a previous run associated with their account, they have to download a zip of all of the files associated with the run, unzip the files, make modifications on their local machine, and upload the new files for a new run. Instead of having to download them, we want the user to be able to modify those files within the browser application. It might also be helpful if a user could choose a previously existing road network from a drop down menu, rather than needing to upload all of the input files which indicate the road network again.

We would also want to implement the ability for a user to do batch simulations. Currently, the frontend only allows one simulation to be started at once. A user might have multiple simulations that they want to test with slightly different parameters. It would be helpful if they didn't need to start one simulation at a time and could instead input all of the information for the various simulations and start them simultaneously. Concurrent simulations are already possible, it's just tedious to start multiple.

We also want to improve upon the current account system. The current system allows for only one administrator account. Also, currently, when a user creates an account, there is no way for them to update their information. In the future, we would like to add the ability for there to be multiple administrators. We would also like for a user to be able to change or reset their password if they forget or if they feel they need a more secure password.

The last item that we would want to improve upon within the current system would be to increase the application's security. Currently, we have implemented some security features, such as confirming that the 'session_token' browser cookie is currently in session and associated with an administrative user before allowing the default executable to be changed. However, we don't verify a session token for several of the API calls. In the future, we would definitely want to add verification for API calls which send information that should be associated with a user. Overall, we would want to make sure that the application is more secure than it is currently.

12. Acknowledgements

We would like to thank our client, Dr. Mohamed Farag, who is in charge of maintaining the Integration Traffic Simulator, for giving us the opportunity to work on this project and for helping us in its development. He can be contacted by email at mmagdy@vt.edu.

13. References

[1] Rakha, Hesham A. *INTEGRATION DYNAMIC TRAFFIC ASSIGNMENT AND SIMULATION SOFTWARE*. https://sites.google.com/a/vt.edu/hrakha/software?authuser=0#h.p_PFphBsaJ3l60.

[2] "Integration Small Version." 2.40, https://sites.google.com/a/vt.edu/hrakha/software?authuser=0#h.p_PFphBsaJ3l60.

[3] *Create React App*. Facebook, https://create-react-app.dev/.

[4] "Installing Node.js via Package Manager." *Node.js*, https://nodejs.org/en/download/package-manager.

[5] "MongoDB Community Server Download." *MongoDB*, https://www.mongodb.com/try/download/community .

[6] "Express 4.x - API Reference." *Express*, https://expressjs.com/en/4x/api.html.

[7] "All download options - Index of /dist/v18.15.0/." *Node.js*, https://nodejs.org/dist/v18.15.0/.

[8] "Hello World Example." *Express*, https://expressjs.com/en/starter/hello-world.html.

[9] "MongoDB Node Driver." *MongoDB*, https://www.mongodb.com/docs/drivers/node/v5.3/.

[10] "Node.js v18.15.0 Documentation." *Node.js*, https://nodejs.org/docs/v18.15.0/api/.