

# Traffic Simulator Input/Output GUI

Emmanuel Aninye, John Beutner, James DeLoach, Kieran Siek, Dhruvil Shah

Client: Mohamed Farag

CS 4624 Multimedia, Hypertext, and Information Access

Virginia Tech, Blacksburg, VA 24061

November 30, 2023

# 1 Abstract

This paper details the development of a user-friendly web-based interface aimed at simplifying the intricate process of configuring the INTEGRATION 2.40 traffic simulation model. The software, while extremely powerful, posed a challenge due to its reliance on multiple, intricately formatted input files that interconnect in ways challenging to validate manually. This project provides a streamlined platform for users to upload edit, manage, and validate these files. Users can upload a predefined set of six files, constituting what is termed an "Input Package," including files such as the Master Control File and Signal Timing Plan File, ensuring backward compatibility. The system validates these Input Packages by individually assessing each field's conditions and cross-validating dependent fields across files, adhering to specifications outlined in the INTEGRATION 2.40 manual. Further, users can seamlessly edit input packages through a user-friendly interface, complete with formatted input files and hint boxes displaying pertinent validation conditions. Real-time feedback on edited fields, represented by error status highlighting, allows users to instantly rectify errors without completing the entire file set before validation. The system also supports the saving of validated files, guiding users to fix validation errors for invalid sets. Additionally, users can conveniently list and manage previously uploaded Input Packages, deciding whether to download compressed files for simulations or initiate further edits. The comprehensive overview includes insights into the system architecture, design choices, and technical implementation. Additionally, the paper offers user and developer manuals, ensuring accessibility for both user groups. Finally, reflections on discoveries and lessons learned during development contribute valuable insights for future projects, underscoring the impact of the web application improved usability of the INTEGRATION

2.40 model. Overall, this paper describes the project's journey, from recognizing the configuration challenges to delivering an effective solution with enhanced usability.

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Problem</b>	<b>7</b>
<b>3</b>	<b>Motivation</b>	<b>8</b>
<b>4</b>	<b>System Requirements</b>	<b>9</b>
4.1	Upload Files . . . . .	9
4.2	Validate Input Packages . . . . .	10
4.3	Field Editing . . . . .	10
4.4	Field Edit Validation . . . . .	10
4.5	Saving Input Packages . . . . .	11
4.6	Listing Input Packages . . . . .	11
<b>5</b>	<b>System Architecture and Design</b>	<b>12</b>
5.1	System Architecture . . . . .	12
5.2	Design . . . . .	12
<b>6</b>	<b>Implementation Details</b>	<b>14</b>
6.1	Frontend Implementation . . . . .	14
6.1.1	Routing . . . . .	14
6.1.2	Communication . . . . .	15
6.1.3	Demand Templating . . . . .	15
6.2	Backend Implementation . . . . .	16
6.2.1	Database Functionality . . . . .	16
6.2.2	Parsing/Validation . . . . .	17

<b>7</b>	<b>Evaluation</b>	<b>19</b>
7.1	Automated Testing . . . . .	19
7.2	Manual Testing . . . . .	19
<b>8</b>	<b>User's Manual</b>	<b>20</b>
8.1	Home Page . . . . .	20
8.2	Upload Page . . . . .	22
8.2.1	Uploading Files . . . . .	23
8.2.2	Editing Fields . . . . .	24
8.2.3	Demand Templates . . . . .	25
8.2.4	Validating Files . . . . .	28
8.2.5	Saving Files . . . . .	30
8.3	Downloading Files . . . . .	30
<b>9</b>	<b>Developer's Manual</b>	<b>32</b>
9.1	Development . . . . .	32
9.2	Deployment . . . . .	33
9.2.1	Traditional Deployment . . . . .	33
9.2.2	Deployment with Docker . . . . .	35
9.2.3	Deployment with Docker Swarm . . . . .	35
9.3	Simulator and Documentation . . . . .	36
9.4	Architecture . . . . .	37
9.5	Frontend . . . . .	37
9.5.1	Home page . . . . .	38
9.5.2	Upload page . . . . .	38
9.5.3	Redux . . . . .	39

9.6	Backend . . . . .	43
9.6.1	Data Storage . . . . .	43
9.6.2	Demand Templating . . . . .	44
9.7	Code Layout . . . . .	44
9.8	Validator Structure . . . . .	50
<b>10</b>	<b>Lessons Learned</b>	<b>52</b>
10.1	Problems Encountered . . . . .	52
10.2	Timeline of Completion . . . . .	53
10.3	Future Work . . . . .	54
<b>11</b>	<b>Acknowledgements</b>	<b>55</b>
<b>12</b>	<b>References</b>	<b>56</b>

## List of Figures

1	System Architecture . . . . .	12
2	Initial Home Page . . . . .	20
3	Home Page with Existing Packages . . . . .	21
4	Copy Dialog . . . . .	21
5	Example Upload Page . . . . .	22
6	Upload File Location . . . . .	23
7	Example File Edit . . . . .	24
8	Demand Vehicle Classes . . . . .	25
9	Demand Templates Section . . . . .	25
10	Single Demand Template . . . . .	26
11	Demand Templates for Lines 1 and 2 . . . . .	27
12	Two Demand Templates for Line 1 . . . . .	28
13	Validate Package Button . . . . .	28
14	Validate File Example . . . . .	29
15	Meta Validation Example . . . . .	29
16	Validation Success . . . . .	30
17	“Leave Site” dialog . . . . .	30

## List of Tables

1	Completion Timeline . . . . .	53
---	-------------------------------	----

## 2 Problem

This project centers around the INTEGRATION 2.40 microscopic traffic simulation software [1]. The Integration model is a traffic simulation model that traces vehicle movements from its origin to destination while giving status updates frequently. The application of this software is to provide estimates of changes that need to be made at any time due to traffic.

The simulation has many different features, the main one being the simulation of traffic flow. For example, it allows for “Steady-state car-following behavior” which means a car decides its speed based on the car in front of it. This is just one example of out of the many different features the simulation has to offer.

The simulation is configured by editing several different text files, which is where the main problem that we seek to address appears. These files are difficult to edit by hand due to the many parameters each of them contains, as well as the fact that the files refer extensively to each other. This issue which is exacerbated by the fact that, for a large simulation, each file might have hundreds or thousands of lines. Furthermore, the files cannot be validated for correctness except by running the simulator.

### 3 Motivation

Developing a web-based interface to manage input files offers a range of benefits that simplify data handling and enhance productivity. Firstly, this approach provides an easy-to-use environment where individuals can effortlessly input the necessary data. This not only makes the task more straightforward but also expedites data entry, saving time and ensuring the accuracy of the information being inputted.

Secondly, the web interface acts as a centralized repository for all files pertaining to each simulation run. This simplifies file management by allowing users to categorize files into logical groups and assign them descriptive labels. It ensures that files are easily locatable and organized, facilitating efficient navigation, review, and editing. This organized approach is particularly advantageous for collaborative projects, enabling teams to collaborate seamlessly and track project progress with ease.

Lastly, the web interface incorporates a built-in validation mechanism that enhances data quality. The validation feature rapidly scans through each file, checking that data values fall within specified ranges and ensuring file consistency. This real-time error detection mechanism eliminates the traditional trial-and-error process, where users typically input files into a simulator, identify errors, and then return to correct them. Instead, users can immediately spot and address errors within the same application where they edit files, streamlining their workflow, and significantly reducing the chances of errors creeping in. In summary, a web-based interface simplifies input and management, streamlines validation processes, and ultimately contributes to more accurate and efficient workflows.

## 4 System Requirements

The requirements for this project are as follows:

1. Users should be able to upload input files as “input packages”
2. Validate sets of input files according to the format specified in the INTEGRATION 2.40 Manual Volume I
3. Allow the user to edit fields of input files in an easy-to-use GUI
4. Validate edited fields before allowing the user to save the files
5. Save the edited input files to the backend
6. List saved input packages from the backend on the frontend

### 4.1 Upload Files

The user should be able to upload a total of 6 files at a time to our application, which includes the Master Control File, Node Characteristic File, Link Characteristic File, Signal Timing Plan File, Origin-Destination Traffic Demands File, and the Incident/Lane Blockages File. These 6 files make up what we call in our application an “Input Package”.

Allowing the user to upload already created files to our application allows for easy backward compatibility as old files will not need to be recreated field-by-field in our application, rather, they can simply upload the files and our parsing mechanism should handle the rest.

## **4.2 Validate Input Packages**

Another important requirement for this project is the ability to validate these sets of files, or “Input Packages”. Each file has many different lines and fields that each have their own validation conditions, generally that they must be numerical within certain ranges as specified in the INTEGRATION 2.40 Manual Volume I. There are also fields that interact and depend across input files, which necessitates the need for a meta-validator that validates conditions between files.

Our program should be able to properly validate each field individually while also cross-validating dependent fields with their respective files.

## **4.3 Field Editing**

Once the Input Packages exist on our application, the user will be able to select the appropriate Input Package that they want to edit and be able to edit a nicely formatted version of the input file. This field editor should also contain hint boxes that display relevant tip information, such as the validation conditions for a particular field, as well as what the field specifies.

## **4.4 Field Edit Validation**

Upon editing the fields of an input file on our application, the user should receive visual feedback indicating whether the value they entered into the field is valid or not. This should be represented through an error status, which can highlight the field red upon validation failure and outline the field green on validation success.

This instant feedback allows the user to constantly keep a valid working set of files as they can instantly see where their errors are and fix them in real-time, without writing the entire set of files before finally validating the set, rather, they can have a running

validation.

#### **4.5 Saving Input Packages**

The user should be able to save the files that they have uploaded once they have confirmed the files are valid. If the user tries to upload a set of invalid files, our application should direct the user to a page where they can begin fixing validation errors.

This will attempt to ensure that our application stores only valid sets of files to prevent the possibility of running a simulation on a set of invalid files, which would fail to run.

#### **4.6 Listing Input Packages**

Once the user has saved an Input Package, or if they are visiting the application after saving Input Packages previously, they should be able to see a list of previously uploaded Input Packages. From this page, the user can decide to either download the Input Package or edit the Input Package. Downloading the Input Package would compress the files into a zip file that the user can use to run simulations while editing the Input Package would go to the previously mentioned Field Editing page.

## 5 System Architecture and Design

### 5.1 System Architecture

The system architecture is as follows:

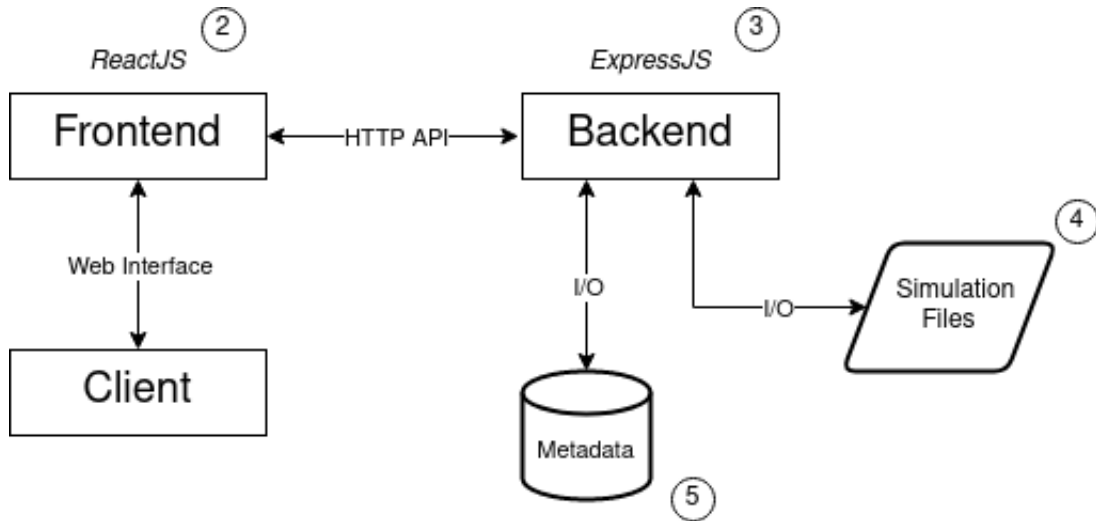


Figure 1: System Architecture

1. The system will be a web interface, which is run on the same machine as the simulation tool.
2. The frontend is written in Typescript and uses the React framework.
3. The backend is written in Typescript as well, and uses the ExpressJS framework.
4. Data files (simulation input files) are stored as raw files on disk.
5. Metadata is stored in an SQLite database.

### 5.2 Design

The system is designed as a web interface to allow for:

- A cross-platform tool, allowing the system to be hosted on most operating systems and consumed from most operating systems.

- Multiple hosting use-cases. For example, the user can run and connect to the webserver locally on a local machine, or host the webserver on a remote machine and connect to it from the client machine.

Typescript is used for both the frontend and backend to allow for more consistent data structures. Since types can be shared between the frontend and backend, this improves consistency of serialization and deserialization of data across the network, reducing bugs and improving reliability.

Data files are stored as plain files on disk because plain files are the format consumed by the simulation tool. Thus, when running the simulation tool, there is no need to export data from the webserver, and the server can just point the tool to the path containing the simulation input files.

Metadata, like package names (packages referring to sets of input files) and modification times are stored in a database for ease of access and consistency.

## 6 Implementation Details

### 6.1 Frontend Implementation

Our frontend is written in typescript using the React framework, in which we leverage the flexibility and built-in form structures of the UI Toolkits that help make developing the application a smoother process.

In our application, we build the frontend and backend together during deployment. The backend serves the frontend's files on the same port as it provides its API. This makes deployment easier: only one service (the backend) needs to be started, and only one port needs to be exposed.

See the attached source code for details.

#### 6.1.1 Routing

Our application uses `react-router-dom` to handle navigation within the app. There are three main endpoints on the frontend application, which are:

- /

This is the home page endpoint, where the user will be brought to when they first access the application. It shows a list of the input packages the user has created, as well as a button to create a new one.

- /upload

This is the upload page endpoint. When users choose to create a new input package, they will be redirected to this page. As described in the User's Manual section, they will then be able to upload existing input files they might have created, or input values manually. They can then validate the package and save it to the backend.

- `/upload/<uuid>`

This endpoint also displays the upload page, but instead of creating a new package it allows editing the existing package with the given UUID.

### 6.1.2 Communication

The frontend communicates to the backend using regular `fetch` requests. The backend has a JSON API, which is specified in detail in a markdown file in the backend directory (`/server/api.md`).

### 6.1.3 Demand Templating

Templating of the demand file (described in further detail in the User’s Manual) is implemented on the frontend. The frontend first loads an input package from the backend (via the “load” API endpoint described below).

Then, since each demand template specifies a range to vary a particular value within, it constructs a separate demand file (and a master file, since it refers to the demand file) for each permutation of every range. For example, if there are two templates which both specify a range from 0 to 1 with a step of 0.1 (for 11 steps total), then the frontend will construct 121 demand files: 0.0 and 0.0, 0.0 and 0.1, ..., 0.0 and 0.9, 0.0 and 1.0, 0.1 and 0.0, 0.1 and 0.1, ..., 1.0 and 1.0. A more complete explanation of this behavior can be found in the User’s Manual.

This is implemented with a loop that maintains a counter for each template. On each iteration it templates the demand and master file, adds them to the zip file being built, and then advances one of the counters, handling rollover and stopping when the last permutation has been constructed.

## 6.2 Backend Implementation

The backend is implemented in Typescript as well and uses the `express.js` backend framework. Our backend is split into two main parts: a file management database with API endpoints and a parsing/validating set of classes.

### 6.2.1 Database Functionality

The database defines these endpoints to provide the frontend the functionality it requires:

- `/api/v1/list`

This endpoint lists all of the input packages that exist in the database, this is used in the main page where the user can select which input package to edit/download.

- `/api/v1/load/:uuid`

This endpoint allows the frontend to load a specific input package (using its UUID). The data can then be displayed to the user, used to construct a zip file, or used to make a copy of the input package.

- `/api/v1/save/:uuid`

This endpoint allows the frontend to save a specific input package. If no package with the given UUID exists then one is created; otherwise it is updated in place. Thus to create new packages, the frontend picks a new UUID to pass to this endpoint, and to edit an existing package the frontend uses the existing package's UUID.

## 6.2.2 Parsing/Validation

The parsers and validators are standalone classes that are not dependent on the server/client functionality. This design choice was to facilitate easy validation on both the frontend and the backend, as they are simply TypeScript classes that perform these parsings/validations.

Each file type has its own validator, where in this case the validator is both the parsing structure and the validation structure. Each validator inherits from an abstract class called `ParserADT<T>`, which contains certain common functionality and makes it easy to create new validators for different types of files. This abstract class is kept in `/shared/parserUtilities.ts`, which also holds validation condition functions that can be combined to create the conditions necessary for a field to be valid (e.g. `isInteger`, `isInRange`, `isLength`, etc.).

For example, the first line in the `MasterFileValidator` has the following format:

```
// Line 1 - File Title
this.lines.push(
  {
    fields: [
      {
        name: "File Title",
        value: fileLines?.[0]?.trim() || "",
        conditions: [
          () => checkLength(this.getField("title").value, 0, 300),
        ]
      }
    ]
  }
);
```

Here we can see that the parser file is a list of lines, where each line is a collection of

fields, and each field has a name, value, and a set of conditions. This structure allows us to easily construct unique conditions for each field while also maintaining the correct structure for outputting the input file.

## 7 Evaluation

### 7.1 Automated Testing

The traffic simulator input file specification provides detailed constraints on the data required in the input files. Additionally, the specification comes with simplified, concrete examples of each input file.

We have implemented a test harness that ensures that expected correct files (from the specification and our of our own making) validate correctly. We also have files that are known to violate the specification and ensure that they fail validation through automated testing.

This application testing was implemented using the Jest framework for the shared validation code. Each of the validators except for the demand file, incident file, and meta-validator has a set of Jest tests to ensure they behave properly in different cases.

### 7.2 Manual Testing

We performed rigorous manual testing of the UI as it was being built. We constructed our own input files using the system to verify they worked. We also tested some sample input files provided with the simulator. Any issues encountered with these files were quickly fixed. Additionally, we performed several careful reviews of the simulation software's manual to ensure our application complied with the specification.

## 8 User's Manual

### 8.1 Home Page

The home page of the application will list all of the input packages that have been saved in the backend. If no packages have been created then the user will only see the option to create a new package. Clicking this button will redirect the user to the upload page, described in further detail in 8.2.

#### Select traffic simulator input package

No input packages found

[New Package](#)

Figure 2: Initial Home Page

If one or more packages have already been created, the user will see a list of them in addition to the option to create a new package.

Clicking a package to expand it reveals options to edit the package, download the package as a zip file, or make a copy of the package.

## Select traffic simulator input package

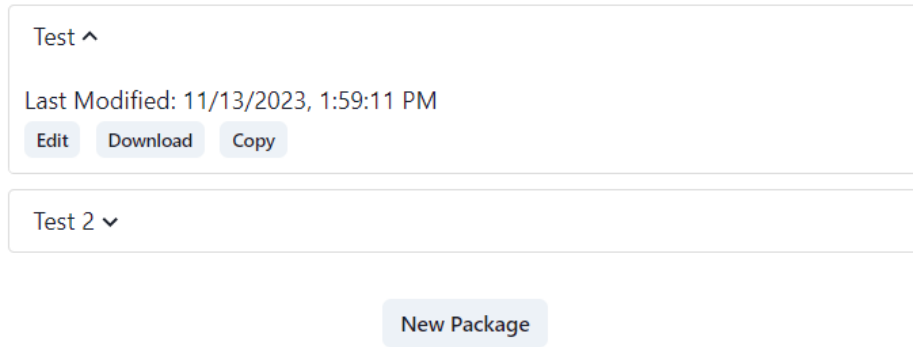


Figure 3: Home Page with Existing Packages

Clicking the Edit button redirects the user to the upload page. The fields will be populated with the data from this package, and users can edit them and save the package as described in Section 8.2. Clicking the Download button causes the system to construct a zip file containing all the input files, possibly performing any templating as described in the “Demand Template” section below. Clicking the Copy button prompts the user to input a new name, then makes an identical copy of the package that has the new name.

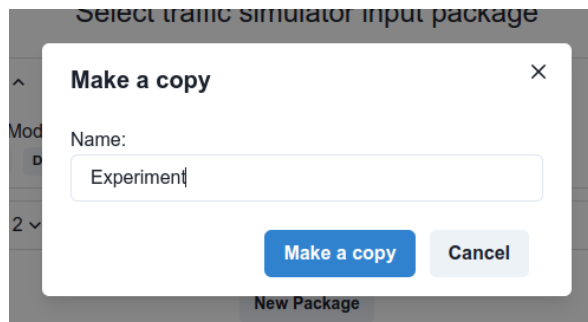


Figure 4: Copy Dialog

## 8.2 Upload Page

The upload page is where users will perform most of their work. Here, a user can upload their files, edit fields, validate files and save packages. Upon being directed to the upload page there are two possible things that will happen:

1. If a user has picked an existing package, the page will populate the forms with the package's data.
2. If a user has clicked "New Package," the page will populate the forms with some generic data.

The screenshot shows the 'Upload Page' interface. At the top, there is a 'Name' input field containing 'Test', with a blue circle '1' next to it. Below this is a horizontal menu with buttons for 'Master File', 'Node File', 'Link File', 'Signal File', 'Demand File', and 'Incidents File'. The 'Master File' button is highlighted in blue, and a blue circle '2' is next to it. Below the menu is a form titled 'Upload Master File' with a blue circle '3' in the top right corner. The form contains a 'Browse...' button followed by the text 'No file selected.'. Below this are several input fields: 'Title:' with the value 'Test'; 'Total Simulation Time:' with the value '3600'; 'Output Frequency (10):' with the value '10'; 'Output Frequency (12-14):' with the value '0'; 'Routing Option:' with a dropdown menu showing '1: Dynamic Traffic Assignment'; 'Pause at end of simulation horizon' with a checked checkbox; and 'Input Directory:' which is partially visible at the bottom.

Figure 5: Example Upload Page

Figure 5 shows an example upload page. Notable UI components are:

1. The *package name* input. Editing this field will change the name of the package

as shown on the home page.

2. The *input file* menu bar. This can be used to cycle through the various input files.
3. The configuration corresponding to the currently selected *input file*.

### 8.2.1 Uploading Files

This page allows a user to upload any files when creating a new package or editing an existing one. For example, if you wanted to run a simulation that is similar to an existing one but has a different incidents file you can make a copy of the package and upload your incidents file and validate your simulation. In Figure 6, the location of the upload button is shown for the Link Characteristic File.

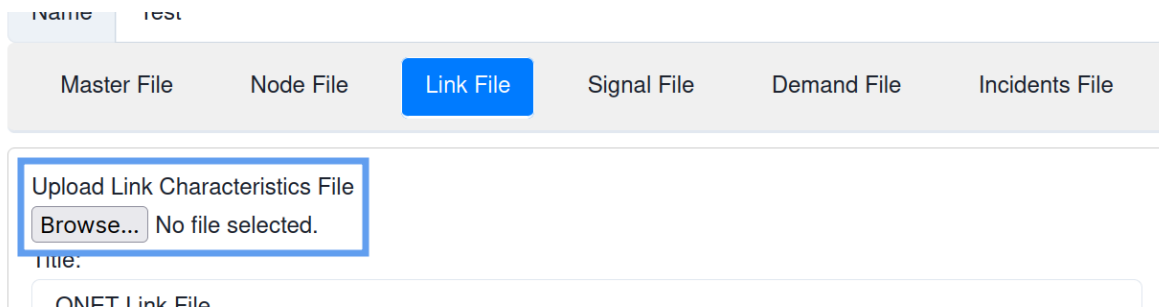


Figure 6: Upload File Location

For any of the files (master file, node file, etc.) uploading a file using the button will overwrite any fields in the frontend with the corresponding values from the uploaded file. The parser makes a best-effort attempt to parse the file and is capable of handling certain types of errors in the input file's format, so corrupted files may still be salvageable by uploading them to the interface.

## 8.2.2 Editing Fields

To make changes to an input file in a new package or an existing package, edit the fields in the configuration box, then save the package to persist changes.

Master File Node File Link File Signal File Demand File Incidents File

Upload Master File  
Browse... No file selected.

Title: 1  
Test

Total Simulation Time: 2  
3600

Output Frequency (10):  
10

Output Frequency (12-14):  
0

Routing Option: 3  
1: Dynamic Traffic Assignment

Pause at end of simulation horizon  
 4

Input Directory:  
./input

Output Directory:  
./output

Figure 7: Example File Edit

The various editing methods (tailored to the specific configuration field) may be (as shown in the Figure 7):

1. Text input
2. Numeric input
3. Selection menu
4. Checkbox

Additionally, some fields (especially in the signal file, due to its more complicated structure) use other input mechanisms.

### 8.2.3 Demand Templates

The system is capable of creating multiple copies of the demand file which allocate different proportions of traffic to different vehicle classes.

CLASS 1	CLASS 2	CLASS 3	CLASS 4	CLASS 5
1	0	0	0	0

Figure 8: Demand Vehicle Classes

Figure 8 shows the portion of one line of the demand file which controls the allocation. As can be seen, there are 5 vehicle classes; the values placed in each of them must add up to 1. The allocation shown in the figure would cause all vehicles to be of class 1. If the values were switched to 0.9, 0.1, 0, 0, 0, then 90% of the vehicles would be class 1 and the rest would be class 2.

To simplify testing with different proportions of each vehicle class, demand templates can be used. Demand templates have a special section on the demand file page, just above the list of demands:

DEMAND LINE	VEHICLE CLASS	START	END	STEP
-------------	---------------	-------	-----	------

**Add**  
Demands

Figure 9: Demand Templates Section

By default no demand templates are created (see Figure 9), so when constructing a zip file to download, the system simply adds each input file directly to the zip file with no modifications.

Demand Templates

DEMAND LINE	VEHICLE CLASS	START	END	STEP	
1	1	0	1	0.1	Remove

Add

Demands

Figure 10: Single Demand Template

If the user creates a demand template, however, when constructing the zip file the system will iterate through the range specified by the template. In the case pictured in Figure 10, the range is 0 to 1 with a step of 0.1, so the system will iterate as follows: 0.0, 0.1, 0.2, ..., 0.9, 1.0. For each value the system will make a copy of the demand file and substitute the value in. In the case shown in Figure 10, the demand template specifies line 1 and class 1, so the system will go to line 1 and replace the allocation to vehicle class 1 with each value in this range. Additionally, since the allocations must all add up to 1, it will then allocate the rest to vehicle class 2.

The system will then add each of these demand files to the zip file (e.g. `demand_0.0.dat`, `demand_0.1.dat`, ..., `demand_1.0.dat`). It will also make a copy of the master file for each value in the range (e.g. `Signal_0.0.int`, `Signal_0.1.int`, ..., `Signal_1.0.int`) since the master file refers to the demand file. Users can then simply pass the master file of their choosing to the simulator to test with a particular allocation.

Users can create multiple demand templates across different lines. For example, in the case pictured in Figure 11 the system will vary the allocation to class 1 on both lines 1 and 2 between 0 and 1 with a step of 0.1. However, they are performed independently. For example, the system will construct a file `demand_0.0_0.0.dat` with class 1 set to 0 on both lines, a file `demand_0.0_0.1.dat` with class 1 set to 0 on line 1 and set to 0.1 on line 2, etc. In this case there are 11 values in the range, so the system will construct

a total of  $11^2 = 121$  demand files and master files.

Demand Templates

DEMAND LINE	VEHICLE CLASS	START	END	STEP	
1	1	0	1	0.1	Remove
2	1	0	1	0.1	Remove

Add Demands

Figure 11: Demand Templates for Lines 1 and 2

Additionally, users can create multiple demand templates which target the same line. For example, in the case pictured in Figure 12 the system will vary the allocation on line 1 to both classes 1 and 2 between 0 and 0.3 with a step of 0.1. Note that these templates are still independent, so the system will construct a file `demand_0.0_0.0.dat` with both classes set to 0.0, a file `demand_0.0_0.1.dat` with the first class set to 0.0 and the second class to 0.1, etc.

For this reason, the “end” values of all the templates which target a particular line cannot sum to a value greater than 1. e.g. in this case, if both ranges went up to 0.6, `demand_0.6_0.6.dat` would have a sum of 1.2 which is disallowed.

The remainder of the allocation is given to the first vehicle class which is not controlled by a template. In this example classes 1 and 2 are templated, so the remainder would be given to class 3. If the second template controlled class 3 instead of class 2, the remainder would instead be given to class 2. It is not possible to template all 5 classes; one class must be left to take the remainder of the allocation.

Demand Templates

DEMAND LINE	VEHICLE CLASS	START	END	STEP	
1	1	0	0.3	0.1	Remove
1	2	0	0.3	0.1	Remove

Add Demands

Figure 12: Two Demand Templates for Line 1

It is possible to mix templates which control different classes on a single line with templates that control other lines; the system is extremely flexible. However, note that each template adds an additional suffix (e.g. `_0.0`, `_0.1`, etc.) to each copy of the demand and master files, and also causes a large jump in the number of demand and master files created. Caution is advised in creating templates to ensure the number of files does not grow excessively and cause resource exhaustion when constructing the zip file. (Note: Since the zip file generation takes place on the frontend, this cannot be used to DOS attack the server, but it is trivial to DOS the client.)

#### 8.2.4 Validating Files

Upon finishing uploading your files and editing your fields you can perform validation on the package by pressing the “Validate Package” button at the bottom, as shown in Figure 13.

---



---



---



---

**Validate Package**

Figure 13: Validate Package Button

The system performs individual validation of each of the files. If there any errors

they will be shown inline, as pictured in Figure 14.

Upload Incident Descriptor File  
 No file chosen  
 Title:

Incidents

INCIDENT RECORD NUMBER	LINK ID	NUMBER OF LANES BLOCKED	SIMULATION TIME SINCE START	SIMULATION TIME TO END	
1	1	1	0	1	<input type="button" value="Remove"/>
2	1	1	0	1	<input type="button" value="Remove"/>
3	1	1	1	0	<input type="button" value="Remove"/>

End Time: Must be after start time

1 more incident

Figure 14: Validate File Example

Once each of the files are individually validated the system will perform meta-validation across files, as pictured in Figure 15.

Last OD pair ID:

Global demand scale factor:

Demand Templates

DEMAND LINE	VEHICLE CLASS	START	END	STEP	
1	1	0	1	0.1	<input type="button" value="Remove"/>

Demands

DEMAND ID	ORIGIN ZONE ID	DESTINATION ZONE ID	DEPARTURE RATE	RANDOM FRACTION OF VEHICLE HEADWAY	BEGIN TIME	END TIME	CLASS 1	CLASS 2	CLASS 3	CLASS 4
1	1	2	2500	1	0	1800	1	0	0	0
2	2	1	1	1	0	1800	1	0	0	0

Destination Zone ID: Node must be of type Destination [1, 2] (Node file)

Origin Zone ID: Node must be of type Origin [1, 3] (Node file)

Figure 15: Meta Validation Example

If the package is successfully validated, an alert shows up to notify the user, as pictured in Figure 16.

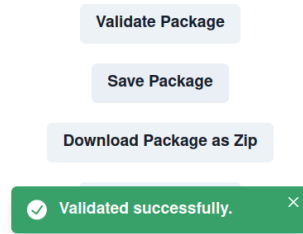


Figure 16: Validation Success

### 8.2.5 Saving Files

If the user clicks “Save Package,” the system will first validate all the files as described above. If all of them validate, the system will then save the package to the backend.

If this is a new package it will be created, and will appear on the home page the next time the user visits it. If the user chose to edit an existing package then it will be updated in place.

If the user tries to close the page or navigate away while there are unsaved changes, the browser will prompt them to make sure they did not simply forget to save their changes, as shown in figure 17.

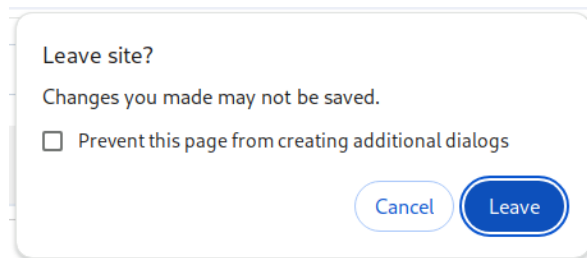


Figure 17: “Leave Site” dialog

## 8.3 Downloading Files

To download a package as a .zip file, there are two options:

1. On the home page, select a package and click “Download.” This will fetch the latest

version of the package from the server, perform any templating of the demand file, and construct a zip file to return to the user.

2. On the upload page, click “Download Package as Zip.” This will validate the package (as described in Section 8.2.4). If validation succeeds, the system will take the values currently shown in the UI, perform any templating of the demand file, and construct a zip file to return to the user.

Note that if the user has been editing the package on the upload page and has not yet saved their changes, the downloaded zip file will contain these unsaved changes, rather than the values that are currently saved on the server.

## 9 Developer's Manual

### 9.1 Development

Since the server's backend uses Node.js, and both the backend and the frontend are built using esbuild, it is first necessary to install Node.js (and the npm package manager) in order to develop the application. These can be installed together from the installers on the Node.js website [2].

Next, `npm i` should be run in the folder containing the project, to install all dependencies. This step needs to be done once at the start, and then again any time the dependencies in `package.json` are updated.

Finally, `npm run watch` and `npm run server:dev` should be run in separate terminals. Both of these should be kept running continuously in order to develop the application. `npm run watch` runs esbuild to rebuild the frontend and backend whenever either changes (it also runs tsc in watch mode to check for Typescript errors, so developers can see if their code contains errors), and `npm run server:dev` runs the backend, restarting it whenever it is rebuilt by esbuild.

Note that `npm run watch:prod` can be run instead of `npm run watch`. This will build the production version instead of the development version. The production builds are smaller and offer better performance, but tools such as the Redux Developer Tools [3] will not work as well for debugging.

The interface should now be accessible by visiting `http://localhost:3000`.

Note that changes to the frontend (and certain changes to the backend) will require reloading the page in order to properly see the updates.

The database (see “Data Storage” below) will be stored in the `local-db` folder within the repository; this directory is `.gitignore-d` to ensure it is not accidentally

committed.

## 9.2 Deployment

This section describes three ways to deploy the project to a server: Traditional deployment as a system service, Docker, or Docker Swarm.

### 9.2.1 Traditional Deployment

To build and deploy the project to a production server:

1. First, note that the project must be built on the same platform (i.e. Linux vs. macOS vs. Windows) as the machine it will be deployed on. This is due to the use of `node-sqlite3` [4], which uses a Node.js native module and must be compiled for a specific platform.

If this is an issue, simply build the project directly on the machine where it will be deployed.

2. Make sure that both the system the project is being built on and the target system of the deploy have Node.js installed.
3. On the build machine, run `npm i` to install all the dependencies.
4. If `npm run watch` is currently running as described above, stop it. Instead, run either `npm run build` or `npm run watch:prod` to build the production version.
5. If the project is being built on a different system than the target system of the deploy, copy the entire contents of the `build` directory to the target system. Note that the directory can be renamed, but the structure of the directory (i.e. separate `client` and `server` directories, each with a few files in them) must be maintained. This is because the server looks for files in specific locations relative to

the `server/index.js` file, and will fail badly if they are missing or in the wrong locations.

6. On the target system, set up a system service to run `node <path to build directory>/server/index.js --local-db <path to database directory>`. The exact mechanism depends on the operating system, but on most Linux distributions the following systemd service can be used:

```
[Unit]
Description=Traffic simulator interface

[Install]
WantedBy=multi-user.target

[Service]
Type=exec
Environment=PORT=3000
ExecStart=node <path to build directory>/server/index.js --local-db <path
to database directory>
User=<user>
```

Note that `<user>` should be replaced with the username of a user to run the service as. The user should have read permissions on the build directory and read-write permissions on the database directory. Additionally, the `PORT` variable can be changed to a value other than 3000 to have the server listen on a different port.

After replacing these values in the above template, log in to the server as root, create a file with these contents at

```
/etc/systemd/system/traffic-simulator-interface.service, run systemctl daemon-reload to force systemd to load the new file, and then run systemctl
```

`enable --now traffic-simulator-interface` to start the service and make it run at startup.

### 9.2.2 Deployment with Docker

The project contains a Dockerfile which can be used to deploy with Docker. Note that Docker must be installed. Instructions for installing Docker Desktop can be found at Docker's Get Started page [5], and instructions for installing Docker Engine (e.g. on a server) can be found at Install Docker Engine [6].

To build the image: `docker build -t traffic-simulator-interface .`

Note that it is possible to build the image on a separate machine, save it as a tar archive (e.g. with `docker image save traffic-simulator-interface >traffic-simulator-interface.tar`), copy the archive to the target system (e.g. with `scp`), and then load the image from the tar archive (e.g. with `docker image load <traffic-simulator-interface.tar`).

To run it: `docker run --rm -p <port>:3000 -v /path/to/database:/db traffic-simulator-interface:latest`

`<port>` should be replaced with the port on the host on which the GUI should be accessible. `/path/to/database` should be replaced with the path to a folder where the backend can store various data.

Note that when launching with `docker run`, the container will not be restarted if the system is rebooted.

### 9.2.3 Deployment with Docker Swarm

The application can also be deployed as a Docker image on Docker Swarm.

First, install Docker as described above. Next, if the system is not already part of a

swarm, run `docker swarm init` to create a single-node swarm.

To deploy the application as a swarm service, after building the image as described above, run `docker service create --replicas 1 --name traffic-simulator-interface --publish <port>:3000 --mount type=bind,source=/path/to/database,destination=/db traffic-simulator-interface:latest`. As when deploying to Docker, `<port>` should be replaced with the port on the host on which the GUI should be accessible, and `/path/to/database` should be replaced with the path to a folder where the backend can store various data. (Note that in a multi-node swarm, this path needs to point to a network filesystem which is mounted on all of the nodes.)

The swarm service can be removed later with `docker service rm traffic-simulator-interface`, or modified with `docker service update <options> traffic-simulator-interface`. Further information can be found on Docker’s website at <https://docs.docker.com/engine/reference/commandline/service/>.

### 9.3 Simulator and Documentation

The simulator, along with some files and documentation, can be downloaded from Hesham A. Rakha’s personal site. [1] (Expand the “Integration Dynamic Traffic Assignment and Simulation Software” section, then click “Download Small Version” at the bottom.)

The file “Integration Manual 1.pdf” contains very useful information about the format of the various input files to the simulator. In brief:

- The master file contains general information about the test, as well as the locations of all the other input files.
- The node characteristic file specifies all the nodes in the network.

- The link characteristic file indicates links between various nodes. Vehicles' motion along these links is controlled by various parameters in the file.
- The signal timing plan file indicates the timing of traffic signals.
- The traffic demand file indicates the origin and destination for various paths along which vehicles will attempt to travel, as well as how often various paths are chosen. An important part of the project involves templating values into the demand file, specifically into the fields which allocate traffic among 5 different vehicle classes. This feature is described in further detail in the user's manual.
- The incidents file indicates when and where incidents will occur that block lanes of traffic.

## 9.4 Architecture

This project is a web application. The backend server is written in Node.js; the frontend is written in JavaScript using React. Since the frontend needs to make API requests to the backend, to simplify matters, the backend serves the compiled frontend code. Thus the user visits one URL (e.g. `http://localhost:3000` for local development), and all the content (including frontend code and API requests) is served from the backend server under this URL.

The API used to communicate between the frontend and backend is specified in the file `server/api.md`.

## 9.5 Frontend

As mentioned before, the front-end uses React which allows us to use components to reduce the amount of code as well as making it easy to pass back and forth information. In order to make the interface appear cleaner we utilized Chakra UI [7]. Chakra UI offers

a plethora of built in components that you can plug and play with and we found this extremely useful while developing. The upload page holds a bulk of the functionality. When navigating to the upload page, it will load in the data from the validators. Each file has its own component for example, the incident file will have a component called `UploadIncidentFileView` and so on. All of these components are stored in the top level component, so that when we save and or validate it is easy to do so.

### **9.5.1 Home page**

The home page (referred to in code as the “list view”) simply makes a request to the frontend to list all the input packages, then displays them in a friendly format. The only other things of note on this page are the Download button (which has a bit of code to actually download the generated zip file, by creating a link and simulating a click) and the Copy button (which prompts the user for a new name, makes a request to the backend to load the package, then makes another request to save it under the new name).

### **9.5.2 Upload page**

The upload page constitutes the bulk of the frontend. To make it more manageable, functionality is divided into multiple components.

The `UploadView` component is rendered directly by the `App` component when the user is on the upload page. If the user selected an existing package, it initially displays a “Loading” screen while it fetches the package from the backend and loads it into the component’s state (some of it is loaded into Redux; see the Redux section for details). Once the package is loaded, it displays a component to edit the name, a tabbed layout with a tab for each file, and the validate/save/download/back buttons at the bottom.

The `UploadView` component passes references to each file's validator to that file's subcomponent, and allows them to replace the validator in-place (e.g. if an entirely new file was uploaded), but it is responsible for actually keeping track of the validators. This is because `UploadView` is responsible for validating, submitting, or constructing a zip file from the package, and this requires it to be able to access all of the validators. (Other state like the name and demand templates is stored in Redux as described below; `UploadView` simply selects the current values from the Redux store when it needs them.)

Special effort is made to minimize renders of `UploadView` itself, since it forces the entire upload page to rerender. For example, this is why the name field is split out to a separate component.

Each input file has a corresponding component to display the portion of the upload page relating to that file (e.g. `UploadNodeFileView`). All of these components have subcomponents to display the individual entries within the file (except for `UploadMasterFileView`, since the master file has no such entries). This avoids rerendering the entire component every time a single value is modified; only the component for that entry in the file is rerendered. Additionally, `UploadDemandFileView` has subcomponents to display the demand templates, which similarly avoids rerendering the entire component when only the demand templates have been modified.

### 9.5.3 Redux

Most of the application's state is stored in the React components (e.g. using `useState`). In particular, the values on the upload page are stored directly in the validators themselves; when the user edits a field the values in the validator are modified. This avoids copying data between the validator and some other location; to validate the package

the validators are simply asked to validate the data they have

However, some state relating to the upload page is stored in a Redux [8] store.

In brief: Redux is designed to contain an application’s entire state (though here we only used it for part of the application for simplicity, as described above). Modern best practice is to split the state up into “slices.” A slice’s state is simply a JavaScript object containing data, and each slice can have one or more “reducers” that can be dispatched to perform modifications to the state (e.g. replacing a particular value). (Note, however, that the Redux store is actually immutable, and code in the reducers which appears to modify the state is actually constructing a new state object, using a library called `immer` – Redux essentially brings functional programming to JavaScript.) *Selectors* are then used to extract particular data from the slice; a selector is simply a function which takes the state as an argument and returns particular values from it. The `react-redux` library provides utility functions for using Redux with React, e.g. the `useSelector` hook, which runs a selector on the store and returns the value returned by the selector. When the store’s state changes, `useSelector` reruns the selector; if the value returned by the selector has changed then the component is rerendered and `useSelector` will return the new value.

The following state is stored in Redux:

- The name of the input package
- The demand templates
- Whether any local changes have been made since the last save (to implement the “Leave site” dialog)
- The lists of errors (individual errors and meta-validation errors) for each line of each file (the name and demand templates also have their own error lists)

The main benefit of using Redux to manage these pieces of the state is that lower-level components can modify them without causing the entire page to re-render, but the code in `UploadView` that validates packages and uploads them to the backend can easily access this state (e.g. to include the name and demand templates in the POST request to the backend, or to update the error lists after performing validation).

Additionally, the lists of errors for each file (except the master file) have special handling to avoid rerenders:

- The components on the upload page which display the errors, rather than selecting the entire list of errors and splitting them up, individually select the list of errors for each line. For example, the node file page creates a child component for each node in the file, and each of those components selects the list of errors for its particular line.
- The selectors which select the list of errors for each line return an empty list if the list of errors has no entry for this line. Additionally, the empty list they return is a “singleton” instance which is created once and reused.
- The reducers which replace the error lists do special work to ensure that as few identity changes occur as possible. For example, if a particular line of the node file has no errors, the singleton empty list described above will be used. If a particular line has the same list of errors as was previously present in the state (this is checked using `lodash.isEqual`), then the previous list is reused. Only if a list of errors is non-empty and has changed will the new list be placed directly into Redux.

The last two points are very important. When using `useSelector` in a React component to select values from the Redux store, any time a value in the store changes the selector is rerun. If an `===` comparison indicates the new value returned by the selector

is different from the previous value, then the component is rerendered and given the new value. Since `===` on an array performs a shallow comparison, techniques like the ones described above are necessary to prevent excessive renders.

Thanks to the special handling described above, however, if the user validates a package, only the components that actually have a user-visible change in the error list will rerender:

- At the start, the list of errors for each file is empty. The selectors will return the singleton empty lists for each line.
- When the user validates a package, lines that do not contain any errors will have the singleton empty list placed into the list of errors at the proper position. The selectors will then return this singleton empty list, which compares as equal and thus does not cause any renders.
- If the user validates a package that contains errors on a particular line, the new list of errors will be placed into the store. The selector for this line will return the new list, which causes a render of the component rendering that line.
- If the user validates the same package without having fixed the errors, the reducer will see that the list is unchanged and will keep the existing list. The selector for this line will return the same list, which compares as equal and thus does not cause any renders.
- If the user fixes some of the errors (or perhaps introduces others) and validates the same package, the reducer will see that the list has changed and place the new list of errors into the store. The selector for this line will return the new list, which causes a render of the component rendering that line.
- If the user fixes all of the errors on that line, the reducer will place the singleton

empty list into the store. The selector for this line will then return this singleton empty list, which causes a rerender. Future re-validations will place the singleton empty list into the store again, which will compare as equal and avoid renders.

Note: The meta-validation error lists are structured slightly differently, with one entry in each line's error list for every field. Thus they required some special handling and cannot uphold quite the same guarantees about no renders. In particular, the first meta-validation will cause most of the page to rerender, because at each line in each file the validator will place a list containing several blank error entries, rather than an actual empty list as was previously returned by the selector. Later meta-validations will generally be faster.

## 9.6 Backend

### 9.6.1 Data Storage

The server stores all of its data within a single folder; the path to this server must be given to the server at startup via the `--db-folder` option. The structure of this folder is as follows:

- `metadata.sqlite` is an SQLite 3 database containing various metadata. Specifically, the `InputPackageMetadata` table stores the UUID, name, and last modification date of each input package, and the `DemandTemplates` table stores the demand templates for each input package.
- `<uuid>/`

For each input package in the database, there is a folder named according that input package's UUID.

This folder contains the master file (named `Signal.int`), as well as all of the input

files. The input files are named and placed in this folder according to the structure defined in the master file (e.g. if the “input directory” field is `input/`, they will all be placed in a subfolder named “input”). Thus the contents of each of these directories would be suitable for immediately running the simulator.

### 9.6.2 Demand Templating

Note: It is recommended to read the section on demand templating in the User’s Manual, which describes in detail the role that this feature serves and how it is expected to behave. In the interests of avoiding needless repetition, this section contains only the additional information useful for development.

The code for validating demand templates can be found in `shared/demandTemplates.ts`. These checks are performed by both the frontend and the backend and are very rigorous.

The code that performs the templating can be found in `client/src/download.ts`. The `buildZipFile` function assembles the zip file, and most of this task is consumed with templating the demand file. If there are multiple templates, every permutation of them is constructed (by looping through as many times as necessary) and used to build an additional demand file, which is then added to the zip file.

Note that in several places while templating, the values are rounded to a fixed number of decimal places. This is because, during initial testing, inaccuracies from floating-point arithmetic caused issues.

## 9.7 Code Layout

- `client/` contains all configuration and code specifically relating to the frontend.
  - `tsconfig.json` contains TypeScript configuration for the client.
  - `esbuild.mjs` contains a script which uses esbuild [9] to build the client. It

takes two options: `--production` to build the production version (more heavily minified, and with a production build of React instead of a development one), and `--watch` to continuously watch for changes and rebuild (the default is to build once and then exit).

Note that this file contains a small custom esbuild plugin that copies `index.html` to the output directory.

– `src/` contains all the TypeScript code for the client.

- \* `index.html` contains the HTML file for the React application. It simply loads the `index.js` and `index.css` files built by esbuild.

Note that those files are compiled files containing all of the CSS/JS, not the specific files named `index.tsx` and `index.css` that can be found in the repository and are described below.

- \* `index.tsx` is the main entry point for the JavaScript code. It creates a React root, sets up `react-router-dom`, sets up a provider for the Redux store, and renders the `App` component inside.

- \* `index.css` contains global styles. It is loaded by `index.tsx`.

- \* `App.tsx` uses `react-router-dom` to render the `ListView` or `UploadView` component (whichever is appropriate based on the URL).

- \* `App.css` contains styling for `App.tsx`.

- \* `ListView.tsx` contains the component to list all input packages from the server.

- \* `ListView.css` contains styling for `ListView.tsx`.

- \* `Nodes.tsx` contains a preliminary version of the node file edit page which was later used to construct `UploadNodeFileView.tsx`.

- \* `Nodes.css` contains styling for `Nodes.tsx`.
  - \* `UploadView.tsx` contains the main component of the upload page, which has the core logic to load and save input packages to the server.
  - \* `UploadView.css` contains styling for `UploadView`.
  - \* `UploadMasterFileView.tsx` contains the portion of the upload page which shows the master file.
  - \* `UploadNodeFileView.tsx` contains the portion of the upload page which shows the node file.
  - \* `UploadLinkFileView.tsx` contains the portion of the upload page which shows the link file.
  - \* `UploadDemandFileView.tsx` contains the portion of the upload page which shows the demand file.
  - \* `UploadSignalFileView.tsx` contains the portion of the upload page which shows the signal file.
  - \* `UploadIncidentsFileView.tsx` contains the portion of the upload page which shows the incidents file.
  - \* `Upload.css` contains styling used across the various “upload” components.
  - \* `download.ts` contains the code to prepare a zip file of an input package for download (i.e. request the package from the server, perform any templating of the demand file, and place the files into a zip file).
  - \* `store.ts` contains the setup for the Redux store described above.
- `server/` contains all configuration and code specifically relating to the backend server.
    - `tsconfig.json` contains TypeScript configuration for the backend server.

- `api.md` contains a specification of the API that the backend server implements.
- `esbuild.mjs` contains a script which uses esbuild [9] to build the server. It takes the same options as `client/esbuild.mjs`.

Note that this file contains custom esbuild plugins which copy the SQLite Node native module and the migration SQL files to the output directory.

- `src/` contains all the TypeScript code for the server.
  - \* `index.ts` is the server's main entry point. It parses the command line arguments, opens the database, and sets up the Express server to serve the frontend and the API.
  - \* `db.ts` contains a class which wraps the SQLite database used to store the metadata, as well as a class which loads stores the texts of each input package to the filesystem.
  - \* `util.ts` contains miscellaneous utilities. Currently that is just an async lock class, and a function to validate and normalize a UUID.
  - \* `migrations/` contains migrations for the SQLite database, in the format accepted by the `sqlite` npm package [10].
    - `001-initial.sql` contains the initial migration, which creates the table to store each input package's metadata.
    - `002-demand-templates.sql` contains a subsequent migration, which creates the table to store each demand template.

- `shared/` contains code that is used by both the client and the server: shared types, parsers, validators, etc.

- `tsconfig.json` contains TypeScript configuration for the shared files.
- `types.ts` contains the Typescript types used to represent each file.

- `demandTemplates.ts` contains the types for the demand templates, as well as some validation code (these are kept separate from the other types and validators since the demand templates are somewhat unique).
- `parserUtilities.ts` contains shared types used by all of the parsers/validators.
- `validators/` contains the validators for each file, as well as the meta-validator.
  - \* `MasterFileValidator.ts` contains the validator for the master file.
  - \* `NodeCharacteristicFileValidator.ts` contains the validator for the node file.
  - \* `LinkCharacteristicFileValidator.ts` contains the validator for the link file.
  - \* `SignalTimingPlanFileValidator.ts` contains the validator for the signal file.
  - \* `ODTrafficDemandFileValidator.ts` contains the validator for the demand file.
  - \* `IncidentDescriptorFileValidator.ts` contains the validator for the incidents file.
  - \* `MetaValidator.ts` contains the meta-validator, which takes as input the other 6 validators and checks conditions which involve cross-checking multiple files.

Note that some of the files (the master file, node file, and signal file) do not reference other files, so the meta-validator will never return any errors for them (and in fact does not have methods to meta-validate them). However they must be passed to the meta-validator because other files refer to

them.

- `tests/` contains tests for the various shared files.
  - \* `TestUtils.ts` has utility functions used by the various tests.
  - \* `MasterFileValidator.test.ts` contains tests for the master file validator.
  - \* `NodeCharacteristicFileValidator.test.ts` contains tests for the node file validator.
  - \* `LinkCharacteristicFileValidator.test.ts` contains tests for the link file validator.
  - \* `SignalTimingPlanFileValidator.test.ts` contains tests for the signal file validator.
- `package.json` contains project metadata such as dependencies, scripts, and some configuration.
- `package-lock.json` is a lockfile automatically generated by npm containing pinned versions of the dependencies.
- `jest.config.js` contains the Jest configuration for the project's tests.
- `tsconfig.json` contains some global TypeScript configuration.
- `Dockerfile` is a Dockerfile that can be used to build a Docker image for the project.
- `.gitignore` is a gitignore file; it contains patterns for files that should be ignored by Git. Mostly this includes built JavaScript and the local database.
- `.dockerignore` contains patterns for files that should be excluded when building a Docker image. Adding unnecessary files here reduces the size of the intermediate “builder” image, as well as allowing Docker to reuse the build cache more often.

## 9.8 Validator Structure

Each validator inherits from an abstract class that performs most of the basic validation subroutines that are needed by each validator. Most of the validators have some type of `mapping` table that contains basic information about where each field is located spatially on the actual formatted input file. Generally, these mappings are two-valued, where the first value is the row and the second value is the column. For example, in the Master File Validator, the `routing_option` field is listed in `field_maps` like so:

```
/**
 * field_maps contains a mapping from every property to its associated field
 */
field_maps: Record<keyof MasterControlFile, [number, number]> = {
  ...
  routing_option: [1, 3],
  ...
};
```

Here we can see that `routing_option` is located in row 2 (with zero-based indexing) and column 4.

Each validator is also structured as an array of `ParserLines` which themselves are arrays of `ParserFields`. This creates the 2D array structure we see the mapping refer to, which is also designed to be identical in structure to how the input files themselves are managed/output.

Each `ParserField` has three attributes, shown below:

```
export interface ParserField {
  name: string;
  value: string;
  conditions: { (): string }[];
```

```
}
```

The name of a `ParserField` is an easy-to-read/diagnosable name for the field and the value is the actual value of the field that will be exported to the final input file. The conditions of a `ParserField` is the set of conditions needed for a `ParserField` to validate successfully. Each function in the conditions array returns an empty string on a successful validation, and a non-empty string on a failure.

To see this in action, we can examine the first line of the Master Control File:

```
// Line 1 – File Title
this.lines.push({
  fields: [
    {
      name: "File Title",
      value: fileLines?.[0]?.trim() || "",
      conditions: [( ) => checkLength(this.getField("title").value, 0, 300)],
    },
  ],
});
```

We see here that the name of the field is "File Title", the value is read in from the file (or replaced by an empty string if it doesn't exist), and the condition is a function that checks the length of the field to ensure it is between 0 and 300 characters long.

Most fields are set up very similarly to this example, with variations on the types of conditions being enforced. Most files also have `ParserLines` that are added/removed based on user input, which is handled slightly differently, but maintains the same structural principles as the example above.

There are many more helper functions in each of the validators, which are documented through Typescript docs-style formatting in the code itself.

## 10 Lessons Learned

### 10.1 Problems Encountered

We learned that client communication is very important. Early on in our project we were confused on the format of the files as well as the general structure of our project. After our client explained this information to us in detail, we realized we should have asked him earlier.

Additionally, once the project progressed we had a meeting with the client in which we realized there had been some miscommunication about an essential feature the client wanted: templating of fields in the demand file. Fortunately we were still moving in the right direction, and adding support for this feature only required relatively small changes to parts of the application.

Along with this, we also learned that it is important to set up clear project milestones in order to give ourselves structure when developing. Since different people would be working on different facets of the project it was important to use a version control system like Git.

While setting up the project we learned that it was important to think carefully on how to setup our project. We had to ask ourselves questions like: How will the front-end communicate with the backend? What information does the frontend need? How will the user interface with the UI, what features do they need? These questions guided us in creating a robust project setup.

When we started linking up the validators in the backend with the frontend editing forms, we noticed a significant drop in performance whenever a single field was being edited. This was due to a couple of factors, one of which being the method by which we exposed the internal input file data to other developers. These problems persisted

especially when our frontend developers would access lists of items in the validation structure, where our implementation initially created a new array representation of the stored data for every individual element access. This made us reconsider portions of the validator’s functionality and led us to implement more efficient accessor methods.

Another performance issue we encountered was with attempting to update the state of our validation structure from user input and representing those changes on the form. Specifically, the mutable nature of the validation structure was not very well suited to React’s state updates, where we resolved this issue by employing a small amount of Redux along with a deferring of validation computation for when the user wants to save or validate their configuration.

## 10.2 Timeline of Completion

August 31, 2023	Technologies and roles decided.
September 7, 2023	Skeleton of project constructed (basic React client, and a basic Express server that served the client).
September 14, 2023	Preliminary UI for node file editing page. Preliminary backend API.
September 25, 2023	Generic parsing and validation system constructed, master file parser/validator implemented.
October 3, 2023	Node file validator implemented. Backend API finalized, and implemented for the master and node files.
October 12, 2023	Link and signal file validators implemented. List page constructed. Download as zip implemented for master and node files.
October 19, 2023	Demand file validator implemented. Various bug fixes, preliminary tests written for master file.
October 28, 2023	Implemented initial upload page with support for for master, node, and demand files. Basic support for demand file templating. Better UI for saving and loading from the backend.
November 2, 2023	Implemented incident file validator. Support for link, signal, and incident files on the upload page and in the backend. Client-side validation UI implemented.
November 11, 2023	Support for meta-validation. Various bug fixes. Project finished.

Table 1: Completion Timeline

### 10.3 Future Work

For anyone planning to continue this project, there are some stretch features that could be implemented. These features include: running the simulator on the server, user authentication, better demand templating, and multiple version support.

A feature that would be beneficial to anyone using this application would be the ability to run the INTEGRATION 2.40 simulation software using the generated input files. This feature would allow users to select an input package that they have created and then run the simulation using that package.

Another feature that would enable the scalability of this application would be user authentication, allowing users to have user-specific input packages that they have created.

Another possible feature would expand the demand templation functionality. While the current demand template functionality is flexible, there are several types of templating it does not support. For example, it is impossible to have the allocations to two different vehicle classes (either on the same line or a different line) be tied to each other, since each template is evaluated independently. If users desire more advanced templating functionality, the system could be modified to support it.

To be more in line with the INTEGRATION 2.40 manual, another feature would be to support all of the different versions that the INTEGRATION 2.40 manual (Volume II) describes. This would enable the application to utilize the full capabilities of the INTEGRATION 2.40 simulation software.

## 11 Acknowledgements

We would like to thank Professor Mohamed Farag, our professor and client, for his assistance in helping us understand how to address this problem. Additionally, we would like to acknowledge Michel Van Aerde and Dr. Hesham Rakha for their work on the INTEGRATION 2.40 model.

## 12 References

- [1] H. A. Rakha, “Software,” last accessed November 28, 2023. <https://sites.google.com/a/vt.edu/hrakha/software>.
- [2] “Node.js,” last accessed November 28, 2023. <https://nodejs.org>.
- [3] “Redux DevTools,” last accessed November 28, 2023. <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknkioeibfkpmmfbljd>.
- [4] Ghost, “node-sqlite3,” last accessed November 28, 2023. <https://www.npmjs.com/package/sqlite3>.
- [5] Docker Inc., “Get Started — Docker,” last accessed November 28, 2023. <https://www.docker.com/get-started/>.
- [6] —, “Install Docker Engine — Docker,” last accessed November 28, 2023. <https://docs.docker.com/engine/install/>.
- [7] Segun Adebayo, “Chakra UI - A simple, modular and accessible component library that gives you the building blocks you need to build your React applications,” last accessed November 28, 2023. <https://chakra-ui.com/>.
- [8] Dan Abramov et al., “Redux,” last accessed November 28, 2023. <https://redux.js.org/>.
- [9] “esbuild - An extremely fast bundler for the web,” last accessed November 28, 2023. <https://esbuild.github.io/>.
- [10] Kriasoft, “SQLite Client for Node.js Apps,” last accessed November 28, 2023. <https://www.npmjs.com/package/sqlite>.