

**OFFLINE TRUCK TRACKER DOCUMENTATION**

Table of Contents

[Purpose of Application 3](#_Toc121814476)

[How to Use 3](#_Toc121814477)

[Programming 4](#_Toc121814478)

[Overall Structure of Application 5](#_Toc121814479)

[Tools to Install 6](#_Toc121814480)

[Frontend 6](#_Toc121814481)

[Backend 8](#_Toc121814482)

[API Layer (trucktracker.api) 8](#_Toc121814483)

[Services Layer (trucktracker.core) 9](#_Toc121814484)

[Data Layer (trucktracker.data) 10](#_Toc121814485)

[Database 11](#_Toc121814486)

[Additional Information 12](#_Toc121814487)

[Things to do 13](#_Toc121814488)

[GitHub 13](#_Toc121814489)

[Helpful References 13](#_Toc121814490)

# Purpose of Application

The purpose of this application is to track the trucks coming from the assembly line. At the time of writing this document, there is no system currently in place to track the trucks once they are off the assembly line. The way trucks were previously tracked was through a spreadsheet that is very prone to user error. See the PowerPoint presentation located in the same directory as this file for more details.

# How to Use

Users must first log in with their Nikola email to be able to utilize this application. Users will be able to do so by clicking the Sign In/Sign Out button located on the top right of the application.

The basic functionality of the application is to be able to drag and drop VIN numbers into the respective buckets. Figure 1 shows what the interface looks like at the time of writing this document.

Diagram

Description automatically generated

**Figure 1: Homepage of the application upon logging in**

Under the “TO BE PLACED” bucket, if there are no trucks available, the dropdown menu shown after clicking the button in the bucket should say “No trucks!”. At the time of writing this document, access to the MES database was not available, so data needs to be manually loaded into the database via SQL. If there are trucks available, however, there will be VIN numbers available. Figure 2 shows what this may look like. The original design of this application involves being able to retrieve VIN numbers as the trucks move off Station 15 in the assembly line, but that functionality was not implemented (more details under the section “Things to do”).

Trucks will also be revealed under the buttons that have a number greater than the value zero, and VIN numbers that appear can be dragged and dropped accordingly. If the number on the button exceeds the value indicated in the title of the bucket, the bucket will be highlighted red to indicate that there is an issue.

Text

Description automatically generated

**Figure 2: TO BE PLACED bucket with an available VIN number. If this VIN number did not exist, the text “No trucks!” would take its place instead. VIN numbers are also displayed in ascending order.**

Once a user successfully drags and drops a VIN number into a bucket, a confirmation pop up will appear and if approved, the movement will be recorded in the database. Users can view this movement by clicking on the text labeled “History” located on the top right of the interface shown in Figure 1. Figure 3 demonstrates what looking up the history of a VIN may look like.

Table

Description automatically generated

**Figure 3: Viewing the history of VIN 153. Notice how the user does not need to input the entire VIN number. At the time of writing, VIN numbers only need to be specified by the last 3 digits, but the full VIN number can be searched if desired.**

Extracting data in a preferred format has not been implemented yet as of writing this document. More on this will be in the “Things to do” section of this document.

# Programming

As a side note, the README.md files will not be included under the list of files found in any directory. These files are just meant to give a brief overview of the portion of the project that the file is meant for on GitHub.

## Overall Structure of Application

As this is a full-stack web application, there are various components that make up how the application works. The components will be broken down into the following list:

* **Frontend (React.js)**

This is the component that directly communicates with the user and is what the user sees when utilizing the application. If bugs are found, the front-end is usually the culprit.

* **Backend (ASP.NET Core utilizing .NET6)**

The backend typically does not need to be touched unless something needs to be implemented in terms of data retrieval. If a bug is found, it will usually have to do with the front-end side of things. It is recommended to utilize an application like Postman to test the backend during development of a new feature.

* + **API Layer (labelled as trucktracker.api)**

This is the layer that makes calls to the Services layer based on what is passed in by the Frontend. It will then return an HTTP response status code to indicate success or failure.

* + **Services Layer (labelled as trucktracker.core)**

This is the layer that utilizes the Data layer in a way that allows for communication between the backend and the database. The overall structure of the functions found in this layer is that if the data retrieval is successful, this layer will return the data back to the API layer. If not, an error will be returned to the in-browser console.

* + **Data Layer (labelled as trucktracker.data)**

This is the layer that is responsible for linking the backend with the SQL Server database. You will find some database triggers in this directory that are necessary for how this application functions; make sure that these triggers are placed in the SQL server that this application is interacting with.

* **Database (SQL Server)**

This is the component that the backend component directly communicates with to retrieve data. The design of the database is located under file name “dbdesign.txt” in the same directory as this file, and it will be noticed that it is just code. It is advised to utilize the website “dbdiagram.io” to see a clearer visual of the database design.

Graphical user interface, application, Teams

Description automatically generated

**Figure 4: Outline of the structure represented visually**

This is just an outline of the structure. An in-depth description of each component will be found in the next sections. It is also worth mentioning that familiarity with Git/GitHub and the general concepts of version control is recommended but is not entirely needed.

## Tools to Install

* Node.js for NPM (<https://nodejs.org/en/>) (If it is still not working post-installation, open a new terminal window and try checking the version number with the following command: **npm -v**)
* SQL server (<https://www.microsoft.com/en-in/sql-server/sql-server-downloads>) (download the free specialized developer edition)
* MSSQL (Microsoft SQL Management Studio) (<https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16>)
* .NET 6 (<https://dotnet.microsoft.com/en-us/download/dotnet/6.0>) (requires restarting PC)
* Visual Studio Code (or any other IDE that is preferred, but VS Code is the main one to be used as it is easy to work with) (<https://code.visualstudio.com/>)

## Frontend

**Prerequisite knowledge:**

* **HTML**
* **CSS**
* **JavaScript (particularly of the React.js library and JSX which is a syntax extension)**

The Frontend is the heart of what the user will see when utilizing the application. The following figures will showcase what the user will typically see.

**To run the React application, execute the following command in the terminal when in the same directory as the project: npm start**

Graphical user interface, text, application

Description automatically generated

**Figure 5: When the user is logged out of their Nikola email, this is the screen that they will see. The user will have to click the “Sign In” button at the top right of the screen shown in this figure.**

The main idea of the application is to have users drag and drop VIN numbers from the “TO BE PLACED” bucket to either the “Dead Truck” or “Align” buckets (or wherever the VIN number needs to be). Most of the functionality has already been discussed in the “How to Use” section, so please refer to that section for more details on how the user is supposed to interact with the application.

The frontend is mainly written in JavaScript utilizing the React.js library and JSX. All the files regarding the frontend will be located under the “src” directory of this project. In this src directory, there are several additional directories that house everything that the application needs to work properly.

1. Components – all the React components go here (more on this in a bit)
2. CSS – all the CSS modules go here, with the React components referencing these modules
3. Images – all images displayed and utilized by the application goes here
4. Pages – all React components that are different “pages” goes here

There are also several JavaScript files located outside of these directories. They will be defined here:

1. App.js – This is the heart of the React application. This does not really need to be touched unless there needs to be another page added or something needs to be implemented regarding authentication.
2. authConfig.js – this is the file responsible for authenticating Nikola employees via their email.
3. Index.js – This is what the React.js library will look for when attempting to run the frontend.
4. Navbar.js – This is a component that appears on the top portion of the application.

Now, as this is a React application, there are various components that are utilized to make up what is shown in Figure 1. These components will be defined here (these definitions are very barebone, more information can be found in the commented code of the project):

1. Bucket.js – responsible for coloring the buckets and displaying the correct name.
2. DropdownItem.js – responsible for displaying any dropdown items if necessary.
3. InfoBox.js – responsible for displaying information in the box next to the “Before Repair” bucket.
4. NumberOfTrucks.js – displays number in the button on the buckets
5. OfflineGrid.js – responsible for laying out the buckets properly in the “Trucks Offline” box.
6. OfflineTrucks.js – this houses the OfflineGrid.js component.
7. OnlineGrid.js – responsible for displaying trucks currently in assembly line as well as the placeholder bucket for when trucks move out of Station 15.
8. OnlineTrucks.js – this houses the Online Grid.js component.
9. SearchTruck.js – responsible for being able to search for trucks and their history. Most of the code and comments for the code can be found in the History.js file itself under the “pages” directory.
10. SignInButton.js – displays the sign in button if the user is signed out.
11. SignOutButton.js – displays the sign out button if the user has signed in.
12. Truck.js – responsible for indicating whether there are trucks in the buckets.
13. TruckTable.js – displays the trucks currently being worked on in the assembly line (not implemented)
14. Welcome.js – displays the first and last name of the user that is logged in on the Navbar component.

Each component may or may not have JavaScript functions depending on what the component is used for. Each function should have a comment block that explains what a particular one does.

Additional components, CSS modules, images, or pages that are implemented should be added to their respective directories highlighted in orange above. Please refer to the files in these directories to understand how the files should be formatted.

The frontend also communicates with the backend by utilizing the Axios library. Axios is utilized for the ease of sending HTTP requests such as GET, POST, or PUT. Again, please refer to a component that has these Axios requests to see how they should be formatted or look at the Axios documentation online for more information.

## Backend

**Prerequisite knowledge:**

* **ASP.NET Core**
* **.NET 6**
* **C# (or at the very least, Object Oriented Programming [OOP] concepts)**
* **Familiar with REST APIs**

The backend is responsible for handling requests that were sent from the frontend. The backend directly communicates with the database to send or retrieve data. This component of the application usually does not need to be touched when debugging unless new features are being developed, and additional information needs to be extracted in a particular way from the database.

**To run the backend, navigate to the backend directory and use the following command:**

**dotnet run --project trucktracker.api/trucktracker.api.csproj**

The backend is split into three different layers, defined and described below:

### API Layer (trucktracker.api)

This is the first layer encountered when data is sent to the backend. If successful, the appropriate HTTP response code will be given. Figure 6 shows a function found in the backend that illustrates what a function is expected to return if successful. As this usually happens in the background, users will typically never see the HTTP response codes. Developers, on the other hand, will be able to see them by utilizing an application called Postman (or any other program that is able to test an API) or by outputting the response of the Axios request to the browser console. Definitions of the HTTP response codes can be found online or as they come when developing and debugging new features.

Text

Description automatically generated

**Figure 6: The “CreatePersonAsync” function found in PersonController.cs should always return status code 201, denoted by the annotation above the function.**

The directories found in the API layer are the following:

1. Controllers – responsible for handling all the calls from the frontend as well as returning the appropriate HTML response code.
2. Models – responsible for providing the skeleton needed for creating or updating data specified in the controllers.
3. Properties – not important to look at but important to the application

The files found in the API layer are the following (all are autogenerated by the project itself):

1. appsettings.Development.json – no need to touch
2. appsettings.json – adjust the “ConnectionStrings” object to accommodate for the usage of the appropriate SQL server.
3. Program.cs – the heart of the backend. Line 26 of this file is important as it specifies the connection to the SQL server, but this will be discussed further in the data layer section
4. trucktracker.api.csproj – utilized to run the backend

### Services Layer (trucktracker.core)

This layer is responsible for handling the data that is sent from the API layer. As mentioned in the outline, it can utilize the data layer to directly communicate with the database. More specifically, it serves as a bridge between the API and the data layer to be sure that data is appropriately being sent or retrieved from the database. Figure 7 shows a function that illustrates this “bridge”:

Text

Description automatically generated

**Figure 7: “AddPersonAsync” function in the services layer. This function retrieves the model from the API layer and passes it to the data layer where the appropriate function is invoked.**

Other than knowing that the services layer acts as a bridge, there is not much to know about this layer. The only other important thing to know is that this being a service is how the API layer can know and draw the connections accordingly. Any additional services need to be defined in the Program.cs file in the API layer.

The directories in the services layer are the following:

1. Interfaces – responsible for acting as a point of reference to the API layer
2. Models – responsible for providing the skeleton that is needed to create the entities required for passing parameters back to the API layer
3. Services – the heart of all the logic associated with this layer

The only file in the services layer is the following:

1. trucktracker.core.csproj – the main link between this layer and the API layer. Not important to touch, but important to the application.

### Data Layer (trucktracker.data)

The data layer is the last of the three layers. This layer is responsible for allowing the services layer to communicate with the database, while also providing the heart of how the database is structured. There are several directories in this layer, and they are the following:

1. DBTriggers – mandatory to put these database triggers into SQL server. These triggers are meant to be for the “TruckHistory” table at the time of writing this. While it is not necessary to update this directory with new triggers, it is extremely recommended to do so during the development phase if utilizing a SQL server on a local computer instead of the SQL server in the Coolidge facility
2. Entities – these are the tables in the database. If a new table were to be created, a new Entity would have to be created and placed into this directory. **If a new Entity were to be created, please be absolutely sure**. A new Entity means a new table in the database, which will affect the entire backend as logic would need to be implemented to accommodate for the new Entity. Otherwise, the newly created Entity will do nothing.
3. Interfaces – similar purpose as the interfaces in the service layer. Allows both the API layer and the service layer to directly reference the repositories declared in this layer. If a new Entity were to be created, an interface needs to be implemented as well.
4. Migrations – the heart of the database. During the initial development of this application, if redesigning the database was necessary, it was as simple as running a few commands to undo the migrations. Once this application is accepting live data, redesigning the database will be much harder. Therefore, it is heavily advised to be 100% sure that the database does not need to be tweaked further once it is time to accept live data. This is not to say that redesigning or adding to the database design is impossible, it will just be much harder and at the time of writing this document, it is unknown how this is done.
5. Repositories – the heart of the data layer. Logic for interacting with the database goes here. If a new Entity were to be created, a repository needs to be implemented as well.

There are also some files in the data layer, and they are the following

1. trucktracker.data.csproj – the main link between this layer and both the API and service layer. Not important to touch but important to the application.
2. TruckTrackerContext.cs – serves as a link between the data layer and the database. Entities are declared here by utilizing the files found in the Entities directory.
3. TruckTrackerContextFactory.cs – provides the connection to the SQL server database.

**\*\*\*IMPORTANT NOTE: it is imperative to remember that to be able to successfully communicate with ANY SQL server, the “optionsBuilder.useSqlServer” line found in TruckTrackerContextFactory.cs must be provided the same string as the one provided in the API layer found in the Program.cs file. Keep in mind that the Program.cs file’s connection string is referenced from the appsettings.json file under “ConnectionStrings”. It is possible to add to the appsettings.json file when necessary.**

Regarding the migrations, it is very important to update the SQL server which the application is currently connected to with the migrations that are within the project files. If this is not done, the application will not know how to store the data. To apply the migrations to the database that the application is connected to, please perform the following command from within the data layer on the console:

**dotnet ef database update**

## Database

**Prerequisite knowledge:**

* **SQL**
* **SQL Server**

There is nothing much to the database other than knowing how to query necessary data when it comes down to doing so when the data is required. For the purposes of this application, the SQL server that is used with the application is the one located at the Coolidge facility (The server is 10.244.150.129). You will need some credentials to log in to this database (contact Dan Grebenisan [dan.grebenisan@nikolamotor.com] for these credentials) but in the meantime, developers can use their local SQL server to do some testing with any necessary dummy data.

As a point of reference, at the time of writing this document, this is the current design of the database:

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 8: Database design for the application on dbdiagram.io**

If viewing a live copy of this is necessary, please reference the “dbdesign.txt” file located in the same location as this document and copy/paste the code found there into dbdiagram.io.

# Additional Information

The IDE (Integrated Development Environment) used to develop this application was Visual Studio Code. As such, there were several extensions utilized to make the development process much more streamlined and convenient. Below is a list of them that were used throughout the development process, although not all of them are necessarily required.

Text

Description automatically generated A screenshot of a computer

Description automatically generated with medium confidence

**Figure 9: List of extensions utilized in Visual Studio Code for the development of the application. Not all of them are necessary, but a majority helps with debugging.**

If it was not already obvious, both the backend and frontend need to be running simultaneously for the application to work. Please review the commands within the relative sections on how to do so.

It is important to note that this is a living document. Any updates to the documentation should be made accordingly as new features are added, or if existing features are removed. As this manual also exists on GitHub, it is imperative to make sure that any local versions of the manual is up to date with the one on GitHub. The manual on GitHub can be thought of as the “master copy”.

# Things to do

There are always improvements that need to be made and implemented for this application. At the time of writing this document, these items will be in a file named “thingstodo.txt” that is in the same directory as this document. Update this as necessary or utilize GitHub’s issue tracker.

# GitHub

Progress on the development of this project is saved on GitHub. As this is a private repository, you will need to ask for permission to access. Please speak to (at the time of writing this document) David Martinez ([david.martinez@nikolamotor.com](mailto:david.martinez@nikolamotor.com)). Regarding accessing and editing the project files, please refer to GitHub’s documentation on how to do so.

If completely new to GitHub, you must create an account. It is imperative to create a new branch from the “master branch” of this repository. The “master branch” is utilized by production (or at least, it will be at the time of writing this), and so if a new feature is being developed, doing so on a new branch would not affect production if something were to go wrong.

# Helpful References

The backbone of this application was based off the following video:

<https://www.youtube.com/watch?v=iYDEEdiLWKI> (Build a full stack web app with React and .NET 5)

If anything in this application appears to be confusing and the documentation does not help for a particular issue, please refer to this video for more information. Note that the application in the video is developed using .NET 5 instead of .NET 6 and at the time of writing this document, .NET 6 was relatively new. Microsoft has plenty of useful documentation for this (as well as for ASP.NET Core), however, so utilize it well here:

<https://learn.microsoft.com/en-us/dotnet/>

Link to dbdiagram.io:

<https://dbdiagram.io/>

Helpful React videos (that helped jumpstart this project)

<https://www.youtube.com/watch?v=hQAHSlTtcmY> (React in 30 mins)

<https://www.youtube.com/watch?v=SLfhMt5OUPI> (Navbar with routing)