| App Documentation

📖 The documentation of the mock program created for the TruckerSystem.

Client Side

Within the client-side subdirectory are all of the front end files.

- index.html
- index.js
- stylesheet.css

These files dictate what is displayed on the web browser, which is seen by the user. The HTML and CSS files create the visual structure and styling of the web page. The JS file allows the web page to be dynamic, and connects to the back end.

index.js

document.addEventListener()

This document starts with an event listener and a callback function. The event that is being watched is the loading of the webpage, specified by `‘DOMContentLoaded’`.

The callback function is activated once the event happens. This function in particular uses the `fetch API`, with a resource request from the server as the argument.

The resource being requested is the API call for getting all of the data in the mock table.

- `'http://localHost:5000/getAll'`

The response is then converted to JSON format, and data field passed into the helper function, `loadHTMLTable()`.

```
document.addEventListener('DOMContentLoaded', function () {
    // Setting the endpoint
    fetch('http://localHost:5000/getAll')
        // Converting response to json format
        .then(response => response.json())
        // Getting data back in json format and logging it to the console. Must access the
data key of the json file
        .then(data => loadHTMLTable(data['data']));
});
```

loadHTMLTable()

This function takes in the data that is returned from the API call, and enters it into the HTML table. It utilizes a query selector to grab the table element. This function is currently unfinished.

```
function loadHTMLTable(data) {
    // Grab the table body
    const table = document.querySelector('table tbody');

    // If there is no data, display that
    if (data.length === 0) {
        table.innerHTML = "<tr><td class='no-data' colspan='5'>No Data</td></tr>";
    }
}
```

Server Side

.env

This file is included so that we can store our database credentials and not have that sensitive information publicly uploaded.

`.env` files are hidden.

When there is a variable precedented by “`process.env.`“ , it is referencing a variable from the .env file.

app.js

This is the application that acts as our server. It is the heart of the application. It currently contains the endpoint routes that are used by the front end file to access/manipulate the database.

Server Startup

At the end of the file is a simple express function call for `listen()`. This function binds and listens for connections on the specified port number (currently 5000).

> 📖 To start the server, run the command
>
> - `node app.js`
>
> You can also use
>
> - `npx nodemon app.js`

```
//-----SERVER STARTUP-----//
app.listen(process.env.PORT, () => {
    console.log("The app is online.");
});
```

dbService.js

This file is the meat of the application. This handles the SQL queries and acts as a class with methods for the app.js file to call.

mysql.createConnection

First and foremost, a connection variable is declared and initiated using the corresponding database information and credentials.

```
const connection = mysql.createConnection({
    host: process.env.HOST,
    port: process.env.DB_PORT,
    user: process.env.USER,
    password: process.env.PASSWORD,
    database: process.env.DATABASE
});
```

The connection variable has a method for starting the connection, which is then called.

There is a callback function included to log an alert of the database connection's state.

```
connection.connect((err) => {
    if (err) {
        console.log(err.message);
    }
    console.log('db ' + connection.state);
});
```

class dbService

This class contains the methods for sending the SQL queries to the database, and handling the returned data.

It is important that only one instance of this class is active at any time, which is why there is a check included in the `getInstance()` method.

The `?:` operator returns a value based on the condition. The condition is listed before the `?` (in this case, the condition is instance ≠ null). If the condition is true, the value on the left side of the `:` is returned. If the condition is false, the value on the right side of the `:` is returned.

To put it simply, if there is an instance created, return it. If not, create a new instance and return that.

```
static getInstance() {
        return instance ? instance : new dbService();
    }
```

getAllData

This is used to send a SQL query to the database that grabs all of the table data.

The response is declared and intiated with a Promise. In simple terms, a promise in javascript is just that. It promises the variable initiated to it that whenever it has the result of the function within the promise, it will hand it over.

When the executor within the promise obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

- `resolve(value)` — if the job is finished successfully, with result `value`.
- `reject(error)` — if an error has occurred, `error` is the error object.

In this case, the executing function is a SQL query being made to the database. The SQL syntax is stored in a string then passed into the connection's `query()` method, along with a callback function, which links back to the promise's (resolve, reject).

After this is completed, the response is returned to the caller of `getAllData()`.

```
async getAllData() {
        try {
            // Create a promise to handle the query. Using a resolve, reject. if query suc
cessful, it will resolve. If not, it will reject and transfer to the catch.
            const response = await new Promise((resolve, reject) => {
                const query = "SELECT * FROM mock_table";
                connection.query(query, (err, results) => {
                    if (err) reject(new Error(err.message));
                    resolve(results);
                });
            });

            return response;

        } catch (error) {
            console.log(error);
        }

    }
```

module.exports

This simple line of code is fairly self-explanatory. It exports the dbService as a module, so that `app.js` can call its methods.

```
//-----MODULE EXPORT-----//
module.exports = dbService;
```

package.json

This is a metadata file for our web application (server). It lists the modules we are using as dependencies.