Vehicle Signal Tracker

Introduction

This documentation provides an overview of the Node.js application developed during the internship. The application consists of two main modules: `sensor.js` and `app.js`. `sensor.js` is responsible for detecting the details of a customer car, such as fuel level, latitude, and longitude. On the other hand, `app.js` functions as a server that allow multiple users to access the application simultaneously. It also provides various notifications and guidence to users based on specific conditions.

Table of Contents

1. Installation

To install and run the Node.js application, follow these steps:

1. Install Node.js on your machine by visiting the official Node.js website (https://nodejs.org) and download the appropriaet executable.
2. Clone the project repository from the provided source control system.
3. Navigate to the project directory using the command line or terminal.
4. Install the project dependencies by running the following command: *npm install*

2. Getting Started

To begin using the programme, complete these steps:

1. Navigate to the project directory in a terminal or command prompt.
2. Run the programme by typing node app.js into the command line. This functions as the server of the application. Here for using async part of the project we can just use the same command to start the app_async.js
3. Once the server is running, open a web browser and enter the URL 'http://localhost:3000' to visit the application.

3. Usage

The Node.js application provides the following functionalities:

- Automatic detection of customers' car details, including fuel level, latitude, and longitude.
- Simultaneous access to the application by multiple users.
- Notifications to users during undesirable circumstances, such as proximity to another car's location.
- Guiding users to the nearest fuel station when the fuel capacity of their car becomes low.
- Providing an UI for customers to login/register their car and get details like the nearest fuel station location on a map

4. Working

In my project, the event loop plays a central role in managing the execution of code and handling asynchronous operations. It constantly checks both the execution stack and the task queue. Whenever I initiate an asynchronous operation, such as a network request or file I/O, it's processed in the background. This means my program can continue executing other tasks without waiting for the asynchronous operation to complete.

One key advantage of Node.js is its event-driven, non-blocking I/O model. Unlike synchronous programming, where operations are executed sequentially, Node.js allows multiple operations to run concurrently. I don't have to wait for one operation to finish before moving on to the next. This feature not only enhances scalability but also makes resource utilization highly efficient.

To handle asynchronous operations effectively, I used various techniques in my project, including callbacks, promises, and async/await syntax. Callbacks allowed me to define functions that execute once asynchronous operations finish. Promises, on the other hand, provided a more structured approach to asynchronous code, making error handling and composition easier. Lastly, async/await syntax simplified my code by making it appear more synchronous while maintaining non-blocking behavior.

In my project's code, I made extensive use of the event loop to manage asynchronous operations. In the app.js file, I set up a server using Express.js and configured middleware for session management. Additionally, I defined routes for login, registration, and simulating sensor data. I also registered event listeners for two crucial events: 'fuel-empty' and 'proximity.' These events are emitted by the sensor object.

Inside the sensor.js file, I created the Sensor class, which extends the EventEmitter class. This class simulates sensor data by updating the fuel level and location of a car at regular intervals. When the fuel level drops to or below 0, it emits the 'fuel-empty' event. Likewise, when the car is

in close proximity to another vehicle, it emits the 'proximity' event. The corresponding event handler functions are defined in the app.js file to take appropriate actions.

By using the event loop and asynchronous functionality, I've been able to achieve considerable performance and responsiveness in my Node.js application. Asynchronous operations enable my event loop to efficiently handle various tasks while waiting for I/O operations to complete. This ensures that system resources are utilized optimally, and my application can handle a significant number of concurrent requests.

5. Async functionalities

Using the async functionalities of NodeJs the original code is modified to add the async functionalities along with some other extra functionality. I have added ExpressJs functionality to display the output to an actual web browser and not just the terminal.

The project consists of three main files:

sensor_async.js: This file contains the code for simulating sensor data, tracking vehicle location, and detecting nearby vehicles.
app_async.js: This file serves as the main application and includes Express.js server configuration, user authentication, and the integration of the sensor functionality.
index.html: This HTML file provides a user interface for displaying sensor data and vehicle locations on a map.
In this documentation, we will provide a detailed explanation of each of these files, along with their functionalities.

Utils.js is a simple utility javascript file which contains functions like calculation distance of the car from the given fuel station and also an rng function to generate unique car id's for newer registration.

sensor_async.js:

This file contains the code for simulating sensor data and tracking vehicle locations. It also includes functionality for detecting nearby vehicles and emitting events when specific conditions are met.

Dependencies:

fs: The Node.js filesystem module for reading and writing files.
calculateEuclideanDistance: A utility function for calculating the Euclidean distance between two sets of coordinates.
EventEmitter: A Node.js module for handling custom events.

Functions:

getRandomFloat(min, max)

This function generates a random floating-point number within a specified range.

findNearestFuelStation(carLocation)

This function reads fuel station locations from a JSON file and finds the nearest fuel station to a given vehicle's location.

Sensor (Class)

simulateSensor(carId): An async method of the Sensor class that simulates sensor data for a specific vehicle (identified by carId). It reads and updates car data stored in a JSON file, calculates fuel consumption, checks for fuel depletion, and emits events when vehicles are in proximity.

Events:

fuel-empty: This event is emitted when a vehicle's fuel level becomes empty. It provides the carId and the nearest fuel station's location.

proximity: This event is emitted when two vehicles are in close proximity (within a specified distance threshold). It provides the carId of both vehicles and the distance between them.

app_async.js:

This file serves as the main application and includes Express.js server configuration, user authentication, and the integration of the sensor functionality.

Dependencies:

express: A popular Node.js web framework for building web applications.
express-session: Middleware for managing user sessions.
fs: The Node.js filesystem module for reading and writing files.
generateCarId: A utility function for generating unique car IDs.
Sensor and findNearestFuelStation: Imported classes and functions from sensor_async.js.
Middleware

requireLogin(req, res, next)

This middleware checks if a user is logged in by verifying the presence of a userId in the session. It is used to protect routes that require authentication.

Functions:

handleFuelEmpty(carId, station): A function that generates a message when a vehicle's fuel level becomes critical.

handleProximity(carId1, carId2, distance): A function that generates a message when two vehicles are in close proximity.

readUserDetails(): An async function that reads user details from a JSON file.

writeUserDetails(userDetails): An async function that writes user details to a JSON file.

Routes:
/login: A route for user login, rendering a login form and handling login requests.

/register: A route for user registration, rendering a registration form and handling registration requests.

/simulateSensorData: A protected route that simulates sensor data for the logged-in user's vehicle. It starts a continuous sensor simulation and emits events when specific conditions are met.

index.html:

This HTML file provides a user interface for displaying sensor data and vehicle locations on a map. It includes scripts for socket.io and Google Maps integration.

Scripts:
socket.io.js: The socket.io library for real-time communication.

Google Maps API: The Google Maps JavaScript API for displaying maps and markers.

Functionality
Real-time sensor data updates are displayed in the output div.

Vehicle locations are plotted on the map using Google Maps.

When two vehicles are in proximity, markers and messages are displayed on the website.

Conclusion
The VehicleSignalTracker project combines sensor data simulation, real-time communication, and user authentication to enhance driver safety and convenience. It allows users to monitor their vehicles, receive critical notifications, and view vehicle locations on a map in real time. This documentation provides an overview of the project's structure, functionality, and dependencies. To get started, follow the setup instructions and run the application.

6. Call Graph

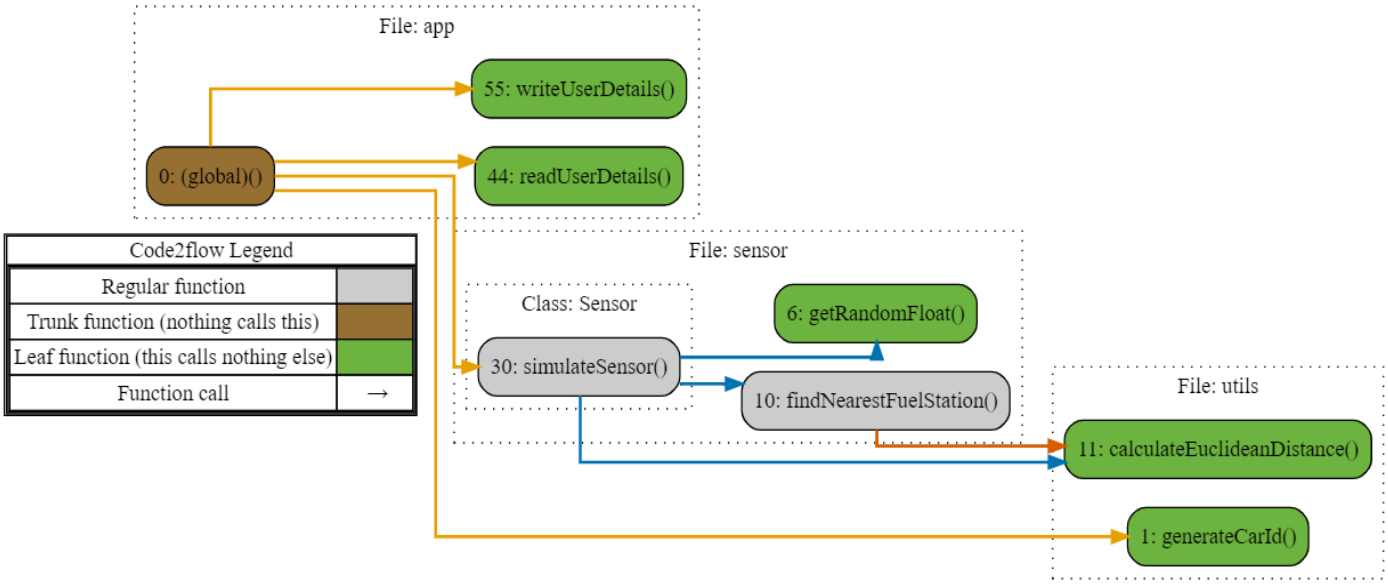

Diagram async:

app.js:

- Functions:
  - readUserDetails(): This function is represented by the node node_78bd2b64. It is a leaf function, indicated by the green color (#6db33f). This means that it does not call any other functions within the codebase.
  - writeUserDetails(): This function is represented by the node node_f828c7e6. Similar to readUserDetails(), it is also a leaf function.
- Dependencies:
  - (global)(): The entry point of the application is represented by the node node_affd3579. It is a trunk function (brown color - #966F33) since it is not called by any other function. It directly calls both readUserDetails() and writeUserDetails().

sensor.js:

- Functions:
  - simulateSensor(): This function is part of the Sensor class and is represented by the node node_bf95a96d. It is a regular function, indicated by the gray color (#cccccc).

        - getRandomFloat(): This function is represented by the node node_730f5086. It is a leaf function and is called multiple times by simulateSensor().
        - findNearestFuelStation(): This function is represented by the node node_8d3f015e. It is called by simulateSensor() and further calls calculateEuclideanDistance().
    - Dependencies:
        - getRandomFloat() is called multiple times by simulateSensor().
        - findNearestFuelStation() is called by simulateSensor().
        - calculateEuclideanDistance() is called by findNearestFuelStation().

utils.js:

- Functions:
    - calculateEuclideanDistance(): This function is represented by the node node_d39f91e6. It is a leaf function and is called by findNearestFuelStation().
    - generateCarId(): This function is represented by the node node_25102017. It is a leaf function and is called by the entry point (global)().
- Dependencies:
    - calculateEuclideanDistance() is called by findNearestFuelStation().
    - generateCarId() is called by the entry point (global)().

The arrows between the nodes represent the flow of control or function calls. Here's a summary of the connections:

- The entry point (global)() calls readUserDetails(), writeUserDetails(), simulateSensor(), and generateCarId().
- simulateSensor() calls getRandomFloat(), findNearestFuelStation(), and calculateEuclideanDistance().
- findNearestFuelStation() calls calculateEuclideanDistance().
- simulateSensor() calls getRandomFloat() multiple times.

The application involves user registration and login functionality, which are implemented using Express routes. The routes /login, /register, and /simulateSensorData are defined and associated with their corresponding HTTP methods (GET and POST). These routes interact with the readUserDetails and writeUserDetails functions to read user details from a JSON file and update it with new registrations.

The server initialization is accomplished through the ‘app.listen’ method, which starts the server and listens on a specific port (in this case, port 3000). Once the server is running, it outputs a log message indicating the successful startup.

The graph provides a visual representation of the code's structure and dependencies. It shows how functions are connected and which functions are called by others. The different colors and

shapes in the graph help distinguish between regular functions, leaf functions, and trunk functions, providing additional context about their roles in the codebase.

7. Conclusion

The VehicleSignalTracker project combines sensor data simulation, real-time communication, and user authentication to enhance driver safety and convenience. It allows users to monitor their vehicles, receive critical notifications, and view vehicle locations on a map in real time. This documentation provides an overview of the project's structure, functionality, and dependencies. To get started, follow the setup instructions and run the application.

Github repository: https://github.com/Shiva-C-qwerty/VehicleSignalTracker

8. References

- Node.js Official Documentation: Node.js Documentation
- Express.js Official Documentation: Express.js Documentation
- MDN Web Docs - Asynchronous Programming in JavaScript: MDN Web Docs - Asynchronous Programming
- Node.js Official Documentation - Understanding the Node.js Event Loop: Node.js Event Loop Guide

These resources provided valuable insights into Node.js development, asynchronous programming, and web application frameworks. I acknowlege that the guidance and knowledge shared by the Node.js and web development communities, were vital in this project.