**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's car, such as fuel level, latitude, and longitude. On the other hand, `app.js` functions as a server that allows multiple users to access the application simultaneously. It also provides various notifications and guidance to users based on specific conditions.

**Table of Contents**

*1. Installation*

*2. Getting Started*

*3. Usage*

*4. Working*

*5. Call Graph*

*6. Conclusion*

*6. References*

**1. Installation**

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

1. Install Node.js on your machine by visiting the official Node.js website (https://nodejs.org) and downloading the appropriate installer for your operating system.

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 start using the ­Node.js application, follow these steps:

1. Ensure that you have completed the installation process mentioned in the previous section.

2. Open a terminal or command prompt and navigate to the project directory.

3. Start the application by executing the following command: *node app.js*

4. Once the server is running, you can access the application by opening a web browser and entering the URL `http://localhost:3000`.

**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.

**4. Working**

Node.js leverages an event loop to efficiently handle asynchronous operations, offering a powerful programming paradigm. Understanding the concepts of the event loop, synchronous, and asynchronous programming is essential when working with Node.js.

The event loop in Node.js is responsible for managing the execution of code and handling asynchronous operations. It continuously checks the execution stack and the task queue. When an asynchronous operation, such as a network request or file I/O, is initiated, it is processed in the background, allowing the program to continue executing other tasks.

In synchronous programming, operations are executed sequentially, and the program waits for each operation to complete before moving on to the next one. This can lead to blocking situations where the program becomes unresponsive. However, Node.js employs an event-driven, non-blocking I/O model. This means that multiple operations can be initiated simultaneously, and the program does not wait for them to complete before moving on. This enables high scalability and efficient resource utilization.

Asynchronous operations in Node.js are typically handled using callbacks, promises, or async/await syntax. Callbacks allow developers to define functions that are executed once an asynchronous operation completes. Promises provide a more structured approach to handling asynchronous code, enabling better error handling and composition. Async/await is a syntax that simplifies working with asynchronous code by using keywords that make it look more synchronous while still being non-blocking.

In the provided code, the event loop is utilized to handle asynchronous operations. In the app.js file, the code sets up a server using Express.js and configures middleware for session management. It defines routes for login, registration, and simulating sensor data. It also registers event listeners for the 'fuel-empty' and 'proximity' events emitted by the sensor object.

The sensor.js file contains the Sensor class, which extends the EventEmitter class. It simulates sensor data by updating the fuel level and location of a car at regular intervals. It emits the 'fuel-empty' event when the fuel level reaches or goes below 0 and the 'proximity' event when the car is in close proximity to another car. These events are handled in the app.js file by the respective event handler functions.

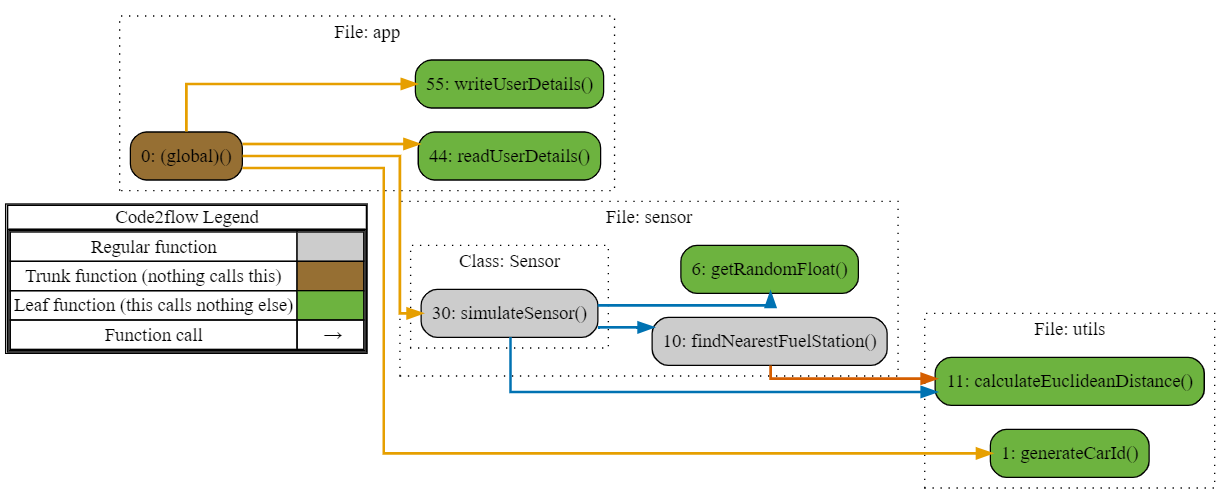
By leveraging the event loop and asynchronous programming, Node.js applications can achieve high performance and responsiveness. Asynchronous operations free up the event loop to handle other tasks while waiting for I/O operations to complete. This enables efficient utilization of system resources and the ability to handle large numbers of concurrent requests.

Understanding the event loop, synchronous, and asynchronous programming concepts is crucial for effectively developing Node.js applications. Developers can create efficient, scalable, and responsive applications by harnessing the power of asynchronous programming and the event loop.

For more in-depth information on the event loop and asynchronous programming in Node.js, refer to the official Node.js documentation and online resources.

**5. Call Graph**

Diagram:



**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.

**6. Conclusion**

In conclusion, the Node.js application developed during the internship provides an efficient way to detect and utilize customer's car details. The combination of the `sensor.js` and `app.js` modules allows for simultaneous access by multiple users. The application's features, such as proximity-based notifications and guidance to the nearest fuel station, enhance the user experience and promote safe and convenient car usage.

**7. References**

Please refer to the following resources for more information on Node.js and related technologies:

- Node.js Official Website:<https://nodejs.org>

- Node.js Documentation: <https://nodejs.org/docs/>

- Express.js Official Website: <https://expressjs.com>

- JavaScript MDN Web Docs: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>