1 Project 1

Due: Sep 15 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. This is followed by a log which should help you understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To ensure that you have set up your VM as specified in the *Course VM* and *Github Setup* documents.
- To get you to write a simple but non-trivial JavaScript program.
- To allow you to familiarize yourself with the programming environment you will be using in this course.
- To make you design an in-memory indexing scheme.

1.2 Background

This project involves reading and querying sensor data. The data is organized in three levels:

1. **Sensor Types**: A sensor type defines the information for a model of a sensor. The information includes the `manufacturer`, `modelNumber`, the kind of `quantity` measured, the `unit` of measurement and the `max` and `min` `limits` of the measurement. Note that any reading outside these `limits` will necessarily be in `error`.

   Each sensor type is assigned a unique `id`.

2. **Sensors**: These are the actual sensors. Each sensor has a specified sensor type. The information includes the `model` (the `id` of the corresponding sensor type), the `period` in seconds after which the sensor makes the next reading, and a `max` and `min` range of `expected` values. Note that any reading outside this `expected` range will constitute an `outOfRange` condition.

   Note that the `expected` range of a sensor will be a sub-interval of the `limits` of the corresponding sensor type.

Each sensor is assigned a unique `id`.

3. **Sensor Data** A stream of the data readings from the sensors. Each reading consists of a `timestamp`'d `value` produced by a sensor identified by its `sensorId`.

   Note that each reading can be classified as being in one of three states:

   `error` The `value` of the reading is outside the limits of the sensor model.

   `outOfRange` The `value` of the reading is outside the range `expected` for the measuring sensor.

   `ok` The value of the reading is within the range of `expected` values.

   It is **guaranteed** that the time between two readings from the same sensor will be an exact multiple of the sensor's `period`.

The data model should become absolutely clear after looking at the *provided data files*.

1.3 Requirements

You must push a `submit/prj1-sol` directory to your github repository such that typing `npm ci` within that directory is sufficient to run the project using `./index.js`.

You are being provided with an `index.js` which provides the required command-line behavior. What you specifically need to do is add code to the provided sensors.js source file as per the requirements in that file.

Additionally, your `submit/prj1-sol` **must** contain a `vm.png` image file to verify that you have set up your VM correctly. Specifically, the image must show a x2go client window running on your VM. The captured x2go window must show a terminal window containing the output of the following commands:

```
$ hostname; hostname -I
$ ls ~/git-repos
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

The behavior of the program is illustrated in this *annotated log*.

1.4 Provided Files

The prj1-sol directory contains a start for your project. It contains the following files:

sensors.js This skeleton file constitutes the guts of your project. You will need to flesh out the skeleton, adding code as per the documentation. You should feel free to add any auxiliary function or method definitions as required.

index.js This file provides the complete command-line behavior which is required by your program. It requires sensors.js. You **must not** modify this file; this ensures that your `Sensors` class meets its specifications and facilitates automated testing by testing only the `Sensors` API.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

Additionally, the *course data directory* contains sensor data files.

1.5 Hints

You should feel free to use any of the functions from the standard library; in particular, functions provided by the Array, String, Number and Math objects useful. You should not need to use any nodejs library functions (except possibly for assert()) or additional npm packages.

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Set up your course VM as per the instructions specified in the *Course VM* and *Github* documents.
2. Read the project requirements thoroughly. Look at the sample log to make sure you understand the necessary behavior.
3. Look into debugging methods for your project. Possibilities include:
   - Logging debugging information onto the terminal using console.log() or console.error().
   - Use the chrome debugger as outlined in this article.
4. Consider what kind of indexing structure you will need to track the sensor information. For each sensor, you will need to be able to access its sensor type and readings easily. Since each sensor is an instance of its sensor type, you can consider using inheritance to model that relationship.

   When designing your indexing structure, you should first think in terms of abstract *associative arrays* and then consider possibilities for implementing these abstract associative arrays in JavaScript. Possibilities:

- Use standard JavaScript objects as associative arrays. Advantages include the ability to using simple `[]`-based access and trivial JSON conversion for subsequent projects. Disadvantages include the lack of any defined order for property keys and the overhead of all the machinery associated with `Object`'s like inheritance.
- Use the relatively new Map addition to the standard library. An advantage of this approach is that it preserves insertion order. It may also be lighter than the `Object` alternative suggested above. Disadvantages include a clumsy API (`get()` and `set()`) and non-trivial work required for JSON conversion which may be useful for subsequent projects.

5. Start your project by creating a `work/prj1-sol` directory in the `i444` or `i544` directory corresponding to your github repository. Change into the newly created `prj1-sol` directory and initialize your project by running `npm init -y`. This will create a `package.json` file; this file should be committed to your repository.

   Run `npm install`. This should create a `package-lock.json`. Be sure to commit this lock file to your github repository too.

6. Capture an image to validate that you have set up your course VM as instructed. Within a terminal window in your `x2go` client window, type in the following commands:

   ```
   $ hostname; hostname -I
   $ ls ~/git-repos
   $ ls ~/cs544
   $ ls ~/i?44
   $ crontab -l | tail -3
   ```

   Use an image capture program on your workstation to capture an image of your `x2go` client window into the file `vm.png` in your `work/prj1-sol` directory. The captured image should clearly shows the terminal window containing the output of the above commands.

7. Copy the provided files into your project directory:

   ```
   $ cp -p $HOME/cs544/projects/prj1/prj1-sol/* .
   ```

   This should copy the `README` template, the `index.js` and the `sensors.js` skeleton file into your project directory.

8. You should be able to run the project, but all commands will return without any results until you replace the @TODO sections with suitable code.

   ```
   $ ./index.js #show usage message
   $ DATA=$HOME/cs544/data/sensors
   $ ./index.js $DATA
   ```

```
add   sensor-type|sensor|sensor-data NAME=VALUE...
clear clear all sensor data
...
load  sensor-type|sensor|sensor-data JSON_FILE
>> find sensor-data sensorId=22
>>
```

9. Replace the `XXX` entries in the `README` template.
10. Commit your project to github:

    ```
    $ git add .
    $ git commit -a -m 'started prj1'
    ```

11. Open the copy of the `sensors.js` file in your project directory. It contains

    (a) A `strict` declaration.

    (b) A skeleton `Sensors class` definition with `@TODO` comments. The comments and `FN_INFOS` table at the end of the file specify the behavior of the methods you need to complete.

    Note that the `class` syntax is a recent addition to JavaScript and is syntactic sugar around JavaScript's more flexible object model. Note that even though the use of this `class` syntax may make students with a background in class-based OOPL's feel more comfortable, there are some differences worth pointing out:

    - No data members can be defined within the `class` body. All "instance variables" must be referenced through the `this` pseudo-variable in both `constructor` and methods. For example, if we want to initialize an instance variable `sensorTypes` in the `constructor()` we may have a statement like:

      ```
      this.sensorTypes = {};
      ```

    - There is no implicit `this`. If an instance method needs to call another instance method of the same object, the method **must** be called through `this`.
    - There is no easy way to get private methods or data. Instead a convention which is often used is to prefix private names with something like a underscore and trust `class` clients to not misuse those names.

    (c) An assignment of the `Sensors class` to `module.exports`. This makes `Sensors` the only declarations available outside the `sensors¬.js` file; all other declarations are local to the file.

    (d) Validation code. This code checks for required values, converts strings to numbers as necessary and inserts default values. The validation is driven by the `FN_INFOS` table towards the end of the file. Note that though this generic validation takes care of most of your validation needs, you will still need some additional validation.

Some points worth making about the provided code:

- All the methods have been declared `async`. This is not needed for this project, but will be necessary for subsequent projects. You can safely ignore the `async` declarations and write code normally.

- For error handling, the convention used is that all user errors should be thrown as lists of objects (for example, see the `throw` at the end of the `validate()` function). If you need to `throw` your own errors, be sure to wrap them in a list. Do so even for a single error:

  ```
  if (!sensorTypes[model]) {
    throw [ `no model ${model} sensor type` ];
  }
  ```

  This allows the calling code to distinguish between exceptions caused by intentional error reporting and those caused inadvertently, allowing the former to be reported as simple error messages whereas the latter will be reported with a full stack trace. This makes it much easier to debug your code when you get unintentional exceptions.

- The code should assume that the property values in the incoming `info` object which is the argument to the methods in `Sensors` **can** be `String`'s, even when the corresponding property is expected to be a number or integer. For example, a sensor reading may come in as:

  ```
  { ...
    value: "123",
    ...
  }
  ```

  This assumption makes it easy to interface with dumb form interfaces (or a dumb cli program for this project).

  Note that it is also permissible for incoming property values to be numbers as in:

  ```
  { ...
    value: 123,
    ...
  }
  ```

  Both of these possibilities are handled correctly by the provided validation code. In both cases, the validation code return value will contain the property as a `Number`.

- In contrast to the previous point, any number-valued properties in the return values of the `Sensors` methods should be JavaScript `Number`'s.

12. Add initialization code to your `constructor()`. It is possible that you will need to execute the same code in your `clear()` method; in that case, you can simply have your constructor call your `clear()` method (using `this.clear()`).

13. Add any utility functions or methods which may be useful. One possiblity would be an `inRange(value, range)` method which returns truthy iff `range.min <= value <= range.max`. This can be useful in classifying the status of a sensor reading.

14. The `add*()` methods should be relatively trivial (YMMV depending on the details of your indexing structure). Implement the `addSensorType()` method. Test using the provided `$DATA/sensor-types.json` file. Using a debugger or `console.log()` verify that you have indeed captured the data. Check for error handling using `$DATA/err-sensor-types.json`.

15. Implement the `addSensor()` method. You will need to validate that the sensor `model` is a valid sensor type id.

16. Implement the `addSensorData()` method. You will need to validate that the id specified by the incoming `sensorId` is a valid sensor id.

17. Start implementing the `findSensorType()` method. There are two cases to consider:

    (a) There is a truthy incoming `id` property. In this case, you should simply return the single matching sensor type or report some kind of not found error.

    (b) There is no truthy incoming `id` property. After the default processing of the `validate()` function, its return value is guaranteed to have `index` and `count` properties. Ignoring possible filter parameters, simply extract sensor types for the specified `index` and `count`. Test.

    Now add support for filter parameters. Test using filters like `manufacturer` or `quantity`.

18. Implement `findSensor()` along the lines of `findSensorType()`. There should be a lot of common functionality between the two; refactor your code so as to share the common functionality.

19. Implement `findSensorData()`. This may be the hardest part of this project, but you should have already thought this through when you designed your indexing structure.

20. Iterate until you meet all requirements.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When it is complete please follow the procedure given in the *github setup directions* to submit your project using the `submit` directory.