1 Project 2

Due: Oct 11 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 give you more experience with JavaScript programming.
- To expose you to mongodb.
- To make you comfortable using nodejs packages and module system.

1.2 Requirements

You must push a `submit/prj2-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.

The API is extremely similar to that for your previous project. The differences include:

- The `Sensors` class must provide persistent storage by using a mongodb database.
- An instance of the `Sensors` class is created externally by calling an `async` **`newSensors()`** factory method. This is for reasons discussed in class.
- All application errors are reported using an array of `AppError` objects. The provided `AppError` class simply wraps an error code and message; the code will make it easy to classify errors in future projects.

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

1.3 Provided Files

The prj2-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.

The provided code does most (**not all**) the validations necessary for this project.

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.

app-error.js A trival class for application errors.

validate.js Validation code from the previous project. Note that it provides generic validation based on types for input parameters. More validation will be necessary, especially for checking that a parameter specifies some other object via its id.

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. It's content is identical to the previous project except that the error data files have been enhanced with objects which shows the errors resulting from references to non-existing objects.

1.4 MongoDB

MongoDB is a popular nosql database. It allows storage of *collections* of *documents* to be accessed by a primary key named `_id`.

In terms of JavaScript, mongodb documents correspond to arbitrarily nested JavaScript `Objects` having a top-level `_id` property which is used as a primary key. If an object does not have an `_id` property, then one will be created with a unique value assigned by mongodb.

- MongoDB provides a basic repertoire of *CRUD Operations*.
- All asynchronous mongo library functions can be called directly using `await`.

- It is important to ensure that all database connections are closed. Otherwise your program will not exit gracefully.

You can play with mongo by starting up a *mongo shell*:

```
$ $ mongo
MongoDB shell version v3.6.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.3
Server has startup warnings:
...
> help
db.help()                    help on db methods
...
exit                         quit the mongo shell
>
```

Since mongodb is available for different languages, make sure that you are looking at the *nodejs documentation*.

- You can get a connection to a mongodb server using the mongo client's asynchronous connect() method.
- Once you have a connection to a server, you can get to a specific database using the synchronous db() method.
- From a database, you can get to a specific collection using the synchronous collection() method.
- Given a collection, you can asynchronously insert into it using the insert*() methods. However, the code for using `insert*()` to do replacements can get complicated. An easier alternative may be to use the asynchronous replaceOne() method with the upsert option.
- Given a collection, you can asynchronously find() a cursor which meets the criteria specified by a query to `find()`. The query can be used to filter the collection; specifically, if the filter specifies an `_id`, then the cursor returned by the `find()` should contain at most one result.
- Given a cursor, you can modify it using the synchronous sort(), skip() and limit() methods.
- Given a cursor, you can get all its results as an array using the asynchronous toArray() method.

1.5 Hints

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

1. Read the project requirements thoroughly. Look at the sample log to make sure you understand the necessary behavior. Review the material covered in class including the user-store example.

2. 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.

     The couple of minutes spent looking at this link and setting up chrome as your debugger for this project will be more than repaid in the time saved adding and removing `console.log()` statements to your code.

3. Consider how you can use mongo to implement this project. to access its sensor type and readings easily. Try to use mongo's facilities as much as possible; for example, instead of writing code for filtering, design your database objects such that you can use the filtering capabilities of mongo's find() method; use the cursor modification methods like skip() and limit() to implement paging within results.

   Since opening a connection to a database is an expensive operation, it is common to open up a connection at the start of a program and hang on to that connection for the duration of the program. It is also important to remember to close the connection before termination of the program.

   [Note that the command-line program for this project performs only a single command for each program run. Nevertheless, the API provided for `Sensors` allows for multiple operations for each instance; hence you should associate the database connection with the instance of `Sensors`.]

4. Start your project by creating a `work/prj2-sol` directory in the `i444` or `i544` directory corresponding to your github repository. Change into the newly created `prj2-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.

5. Install the mongodb client library using `npm install mongodb`. It will install the library and its dependencies into a `node_modules` directory created within your current directory. It will also create a `package-¬ lock.json` which must be committed into your git repository.

   The created `node_modules` directory should **not** be committed to git. You can ensure that it will not be committed by simply mentioning it in a `.gitignore` file. You should have already taken care of this if you followed the directions provided when setting up github. If you have not done so, please add a line containing simply `node_modules` to a `.gitignore` file at the top-level of your `i444` or `i544` github project.

6. Copy the provided files into your project directory:

```
$ cp -pr $HOME/cs544/projects/prj2/prj2-sol/* .
```

   This should copy in the `README` template, the `index.js`, the `sensors.js` skeleton file and the utility files `app-error.js` and `validate.js` into your project directory.

7. 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.

   The provided code does have sufficient functionality to get a usage message:

```
$ ./index.js
usage: index.js MONGO_DB_URL CMD [ARGS...]
Command CMD can be
add   sensor-type|sensor|sensor-data NAME=VALUE...
clear clear all sensor data
find  sensor-type|sensor|sensor-data [NAME=VALUE...]
help  output this message
load  sensor-type|sensor|sensor-data JSON_FILE...
```

   or do some simple validations:

```
$ ./index.js GARBAGE_URL clear _id=1
sorry; clear does not accept any arguments
$ ./index.js GARBAGE_URL find sensor-type _id=1
RESERVED: the _id parameter is reserved for internal use only
```

8. Replace the `XXX` entries in the `README` template.

9. Commit your project to github:

```
$ git add .
$ git commit -a -m 'started prj2'
```

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

    (a) A `strict` declaration followed by initial imports. The local imports include the `AppError` wrapper class as well as a `validate()` module which contains the generic validation code from the previous project.

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

    (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) Options that you should use when creating a connection to a mongo db.

(e) A trivial utility function.

Some points worth making about the provided code:

- All the methods have been declared `async`. This allows you to `await` asynchronous calls to mongo library functions.
- For error handling, the convention used is that all user errors should be thrown as lists of `AppError` (for example, see the `throw`'s within the validate module). If you need to `throw` your own errors, be sure to wrap them into `AppError` objects in a list. Wrap in the list even for a single error:

```
if (/* no sensor-type for specified sensor model */) {
  const err = `no model ${model} sensor type`;
  throw [ new AppError('X_ID', err) ];
}
```

  The two levels of wrapping achieves the following:

  - Wrapping in the `AppError` allows an error object to contain a `code` which can be used subsequently to classify the error.
  - Wrapping the errors into a list allows reporting multiple errors in a single API call. It also allows the calling code to distinguish between exceptions caused by intentional error reporting and those caused inadvertently; the former can be reported as simple error messages whereas the latter will be reported using a full stack trace. This makes it much easier to debug your code when you get unintentional exceptions.

11. Start by implementing the factory method for `newSensors()`.
    - Validate the parameters to the method. The `mongoDbUrl` must be a valid URL of the form `mongodb://HOST:PORT/DB`. If invalid, return a suitable error.
    - If the parameters are valid, connect to the database (note that you will need to split `mongoDbUrl` into the base part and the database name).
    - Finally, synchronously call the `constructor()`. The constructor should cache the database client connection in the instance and set up instance variables for the database collections you are using.

    [An instance of a `Sensors` should contain a database connection, but obtaining a database connection is an asynchronous operation. Since it is impossible to have an *asynchronous constructor*, an `async` factory method provides a workaround].
12. Implement the `close()` method for `Sensors`. You simply need to `await` a `close()` on your database `client`.

13. The code for adding a sensor-type, sensor or sensor-data is likely to be very similar. Factor out this commonality into a "private" method (there are no "private" methods in JavaScript; a common convention is to indicate that a method is intended to be private by having its name start with a _.).
14. As for the `add*()` methods, many of the `find*()` methods will be very similar. Once again, factor out this commonality into a "private" method.
15. Implement the `add*()` methods as wrappers which simply call the private methods but also take care of any validations not handled by the `validate()` function.
16. Implement the `findSensorType()` and `findSensor()` methods. They should largely be wrappers calling your private methods.
17. Implement the `findSensorData()` method. It too will call your private methods, but because of the non-standard use of `timestamp`'s for paging through the data, this function is likely to contain non-trivial logic.
18. 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.