1 Project 3

Due: Oct 30 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 introduce you to building web services.
- To give you some familiarity with using the express.js web framework.

1.2 Requirements

You must push a `submit/prj3-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 to start the program:

```
$ ./index.js PORT MONGO_DB_URL \
   [ SENSOR_TYPES_JSON SENSORS_JSON SENSOR_DATA_JSON ]
```

where the arguments are:

`PORT` This required argument is the port at which the program listens for web requests.

`MONGO_DB_URL` This required argument is the url specifying the mongo database to be used by the program.

`SENSOR_TYPES_JSON SENSORS_JSON SENSOR_DATA_JSON` These three optional arguments specify the path to files containing JSON data representing the sensor types, sensors and sensor data respectively. If present, the data in the specified database is replaced by the data in these files.

What you specifically need to do is add code to the provided sensors-ws.js such that your server recognizes the following 9 HTTP requests:

`GET /sensor-types` and `GET /sensors` The above URLs may include an optional query string. The following meta-parameters should be recognized within the query string:

- `_index` The starting index of the first returned data item (default 0).
- `_count` Maximum number of data items to be returned (default 5).
- `_doDetail` This query parameter is meaningful only for the `/sensors` URL. If specified, then each returned sensor data item should have a `sensorType` property giving information about its sensor type.

Any additional query parameters should act as filter parameters to filter the returned data items. Specifically, a returned data item must have properties with values identical to those specified for the filter query parameters.

A successful request at these urls should return a JSON representation of a JavaScript object having the following fields:

- `data` This property is required. If the URL is `/sensor-types`, then `data` should give a list of sensor-type items. If the URL is `/sensors` then `data` should give a list of sensor-type items. Exactly what goes into a data item depends on the properties stored for a sensor-type or sensor, but minimally they should include the following properties:
  - `id` A unique ID for the sensor-type or sensor. The data items should be sorted in ascending lexicographical order by `id`.
  - `self` This should specify a URL which will return a result containing that exact item.
- `self` This property is required and should specify a URL which will return exactly the same results.
- `next` This property is optional, but should be specified when there is a possibility for more results. If specified, it should specify a URL which will return the next batch of results for the same query.
- `prev` This property is optional, but should be specified when there is a possibility for earlier results. If specified, it should specify a URL which will return the previous batch of results for the same query.

The `next` and `prev` URLs allow a client to go back-and-forth within the returned results.

If there are no results for a particular combination of filter parameters, then except for one case, the request should succeed with the `data` property returned as an empty list. The exceptional case is when an `id` is specified as a query parameter and there are no results. In that case, the request should fail with a `404 NOT_FOUND` HTTP status code and the returned JSON object should contain a suitable `errors` list as defined below.

`GET /sensor-types/`*ID* and `GET /sensors/`*ID* The behavior of these requests should be similar to the earlier requests except that a success result should contain exactly one data item with an `id` property matching *ID*. The entire result as well as the single data item should both have `self` URLs. There should not be any `next` and `prev` URLs as they do not make sense when only a single data item is expected.

When there are no results, these requests should fail with a `404 NOT_FOUND` HTTP status code and the returned JSON object should contain a suitable `errors` list as defined below.

`POST /sensor-types` and `POST /sensors` These requests should be made with a JSON body specifying a sensor-type or sensor object which is to be added or replaced. The exact properties in the object being `POST`ed will depend on the *data model*, but should minimally include an `id` property.

A successful result should result in a `201 CREATED` HTTP response. The response headers must include a `Location` header specifying a URL at which the newly created or replaced resource may be accessed.

`GET /sensor-data/`*SENSOR_ID* The above URL may include an optional query string. The following parameters should be recognized within the query string:

`_count` This meta-parameter gives the maximum number of data items to be returned (default 5).

`_doDetail` If this meta-parameter is specified, then the returned object should have `sensor` and `sensorType` properties giving details about the sensor and sensor-type producing the sensor-data.

`timestamp` This query parameter specifies an upper-bound on the timestamp of returned sensor-data items. If not specified, then it will default to the latest timestamp associated with the sensor specified by *SENSOR_ID*.

`statuses` This can be specified as `ok`, `outOfRange`, `error` or `all` with meaning as specified in *Project 1*. It should default to `ok`.

Other query parameters serve as filter query parameters to filter the returned data items. Specifically, a returned data item must have properties with values identical to those specified for the filter query parameters.

A successful request at these urls should return a JSON representation of a JavaScript object having the following fields:

`data` This property is required and will be a list of sensor-data items for the specified *SENSOR_ID*. Each item must have a `timestamp` property giving the time of the reading with the items sorted in descending order by `timestamp`, a `value` property giving the value of the sensor reading and a `status` property specifying whether the reading was `ok`, `outOfRange` or `error` as specified in *Project 1*. Additionally, each item must have a `self` URL.

`self` This property is required and should specify a URL which will return exactly the same results.

If there are no results for a particular combination of filter parameters, then the request should succeed with the `data` property returned as an empty list. OTOH, if the *SENSOR_ID* is invalid, then the request should fail with a `404 NOT_FOUND` HTTP status code and the returned JSON object should contain a suitable `errors` list as defined below.

`GET /sensor-data/`***SENSOR_ID***`/`***TIMESTAMP*** The behavior of this request should be very similar to the earlier request except that a success result should contain exactly one data item with a `timestamp` property matching *TIMESTAMP*. The returned sensor-data can have any `status`; i.e., unlike the earlier request `statuses` defaults to `all`.

The entire result as well as the single data item should both have `self` URLs.

When there are no results because of a non-existent *SENSOR_ID* or *TIMESTAMP* this request should fail with a `404 NOT_FOUND` HTTP status code and the returned JSON object should contain a suitable `errors` list as defined below.

`POST /sensor-data/`***SENSOR_ID*** This requests should be made with a JSON body specifying a reading for sensor *SENSOR_ID* which is to be added or replaced. The object being `POST`ed must include a `timestamp` and `value` properties which meet the restrictions of the data model.

A successful result should result in a `201 CREATED` HTTP response without any body. The response headers must include a `Location` header specifying a URL at which the newly created or replaced resource may be accessed.

Errors: If a request fails, then the returned response should have a suitable HTTP status code. The body for the error response should be the JSON representation of an object containing an `errors` property which should specify a non-empty list of error objects. Each error object should have `code` and `message` properties, where `code` is a brief code describing the error and `message` is a possibly context-dependent description of the error.

If the client accesses a URL different from any of those documented above, then the default error behavior provided by the web server framework is acceptable.

Additionally, the program should also log a suitable internal error object on standard error; this error object can contain internal details not exposed to the web service.

The behavior of the program is illustrated in this *annotated log*. Additionally, a working version of the project is available at *<http://zdu.binghamton.edu:2345>* (note that this URL will only work from within the campus network).

1.3 Provided Files

The prj3-sol directory contains a start for your project. It contains the following files:

sensors-ws.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.

sensors.js This file is adapted from the solution to *Project 2*. The only change is to have the `findSensorTypes()` and `findSensors()` return a `previousIndex` property to allow scrolling backwards in the results.

You should feel free to replace this file with that from your solution to *Project 2* with suitable modifications.

index.js This file provides the complete command-line behavior which is required by your program. It requires sensors.js and sensors-ws.js. You **must not** modify this file.

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

validate.js Validation code from the previous project with a bug fix which ensures that when `findSensor()` is provided with a `period` search parameter, then that `period` is treated as an integer.

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.

1.4 Hints

You will need to use some kind of web service client for testing the services you are implementing. The recommended command-line client is curl; it's use is illustrated by the sample log. There are no strong recommendations for GUI clients; possibilities include restlet and yarc.

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-ws example and the express.js documentation.

   Based on the above, think seriously about how you would implement the project. Even though there are 9 separate methods, there is a lot of commonality between the methods. It is possible to factor out this commanality so that you need far fewer than 9 separate handlers. The fact that JavaScript allows first-class functions makes the factoring out very easy.

2. Start your project by copying in the provided files into your `work` directory:

   ```
   $ cd ~/i?44/work  #change into your work directory
   $ cp -pr $HOME/cs544/projects/prj3/prj3-sol .
   ```

   This should copy in the `README` template, the `index.js`, the `sensors.js` file and the utility files `app-error.js` and `validate.js` as well as the `sensors-ws.js` skeleton file into a newly created `work/prj3-sol` directory.

3. Change into the newly created `prj3-sol` directory and initialize your project by running `npm init`. Specifically:

   ```
   $ cd prj3-sol
   $ npm init -y
   ```

   This will create a `package.json` file; this file should be committed to your repository.

4. Install necessary external packages `cors`, `mongodb`, `express` and `body-¬parser`:

   ```
   $ npm install mongodb cors express body-parser
   ```

   The libraries and dependencies will be installed 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 `node_modules` directory should **not** be committed to git.

5. You should be able to run the project enough to get a usage message:

   ```
   $ ./index.js
   usage: index.js PORT MONGO_DB_URL \
     [ SENSOR_TYPES_JSON SENSORS_JSON SENSOR_DATA_JSON ]
   ```

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

7. Commit your project to github:

```
$ git add .
$ git commit -a -m 'started prj3'
```

8. Unlike your previous projects, the provided `sensors-ws.js` skeleton file is extremely bare. To start with, work off the user-ws example covered in class to create an `express app` and have it listen on the specified `port` and print out a message with the `listen` handler.

9. Again, working off the user-ws example, start adding routing to your code. Add a route to a handler for `GET /sensor-type` and use `curl` or a REST client to hit that handler. Use `console.log()` or a debugger to ensure that the handler gets hit.

10. Add code to implement the full functionality for `GET /sensor-types`. Most of the work can be done by the `sensors.findSensorType()` method. You will just need to translate the web inputs (query parameters) to the input parameters to the method and the result of the method to a JSON web response. You will also need to translate any exceptions thrown by the method to suitable HTTP status codes and a JSON error response. For now, you can ignore the HATEOAS `self`, `next` and `prev` properties.

    For error handling, catch any errors which are thrown by the underlying code, log them on standard error and massage them errors into the JSON required by the project specifications. Again, use the error handling in the user-ws example as a guide.

    Test using `curl` or any REST client until you have this working reasonably well.

11. Consider implementing `GET /sensors`. You should see that this is very similar to `GET /sensor-types`. Refactor your sensor-types code so that it can handle both sensor-types and sensors.

12. Consider implementing `GET /sensor-types/`*ID* and `GET /sensors/`*ID*. Again, it may be possible to refactor your earlier handler to handle these too.

13. Implement the `POST /sensor-types/` and `POST /sensors/` methods. Again, most of the work will be done by the `sensors` object. What you will need is to pull the passed-in data from the web request body and pass it into the corresponding `sensor` method.

14. Implement the methods for sensor-data. These will be very similar to your earlier methods.

15. Add support for the HATEOAS `self`, `next` and `prev` properties.

16. 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.