1 Project 4

Due: Nov 20 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 using web services.
- To expose you to a templating system like mustache.
- To give you further experience with using the express.js web framework.

1.2 Requirements

You must push a `submit/prj4-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 SENSORS_WS_BASE_URL
```

where the arguments are:

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

`SENSORS_WS_BASE_URL` This required argument is the url specifying web URL at which an instance of the sensors web service from *Project 3* should be running.

[In order to facilitate automated testing, all UI controls and result values are required to be labelled by HTML class names starting with the prefix `tst-`. In the requirements which follow, a mention of a name starting with `tst-` means that the relevant control or value must be labelled by that `class`.]

What you specifically need to do is add code to the provided sensors.js such that your server displays the following pages:

Home Page at URL / This page should display links to the following pages:

- **Sensor Types Search Page** `tst-sensor-types-search`.
- **Sensor Type Add Page** `tst-sensor-types-add`.
- **Sensors Search Page** `tst-sensors-search`.
- **Sensors Add Page** `tst-sensors-add`.

Sensor Types Search Page This page must be identified by having any element with class `tst-sensor-types-search-page`. It should have two sections:

- A search form for sensor types allowing searching by all sensor type fields excepts the limits fields. (see below).
- A results section showing the results delivered by the underlying web service from the last search. Initially, the page should show the results of doing a sensor types search with all fields empty.

Sensor Types Add Page This page must be identified by having any element with class `tst-sensor-types-add-page`. It should contain a form allowing the user to specify values for all sensor type fields (see below). Successful submission of this form should create or update a sensor type using the underlying web service and transfer control to the sensor types search page set up to show the results for the sensor type just created or updated.

Sensors Search Page This page must be identified by having any element with class `tst-sensors-search-page`. It should have two sections:

- A search form for sensor types allowing searching by all sensor type fields excepts the expected fields. (see below).
- A results section showing the results delivered by the underlying web service from the last search. Initially, the page should show the results of doing a sensors search with all fields empty.

Sensors Add Page This page must be identified by having any element with class `tst-sensors-add-page`. It should contain a form allowing the user to specify values for all sensor fields (see below). Successful submission of this form should create or update a sensor using the underlying web service and transfer control to the sensor search page set up to show the results for the sensor just created or updated.

The results section shown for the above search pages should contain a maximum of 5 sets of results. Additionally, each set of results should show links for the next (`tst-next`) or previous (`tst-prev`) set of results, if applicable.

All of the above pages except the home page must contain a footer having links to the above pages with classes `tst-home`, `tst-sensor-types-search`, `tst-¬ sensor-types-add`, `tst-sensors-search` and `tst-sensors-add` respectively.

1.2.1 Sensor Type Fields

Sensor types have the following fields:

Sensor Type ID `tst-sensor-type-id` Uniquely identifying a sensor type. Can only contain alphanumerics, `-` or `_` characters.

Manufacturer `tst-manufacturer` Can contain only `-`, `'`, space or alphabetic characters.

Model Number `tst-model-number` Can contain only `-`, `'`, space or alphanumeric characters.

Quantity `tst-quantity` Can only have internal values `temperature`, `pressure`, `flow` or `humidity`.

Minimum Limit `tst-limits-min` A number.

Maximum Limit `tst-limits-max` A number.

Note that there is no field specifying the units for the limits. Instead, the application should ensure that an appropriate unit is sent to the web service based on the specified quantity: `gpm` for `flow`, `%` for `humidity`, `PSI` for `pressure` and `C` for `temperature`.

1.2.2 Sensor Fields

Sensors have the following fields:

Sensor ID `tst-sensor-id` Uniquely identifying a sensor. Can only contain alphanumerics, `-` or `_` characters.

Model `tst-model` Can contain only alphanumerics, `-` or `_` characters.

Period `tst-period` Must be an integer.

Expected Minimum `tst-expected-min` A number.

Expected Maximum `tst-expected-max` A number.

1.2.3 Error Handling

Invalid form input should result in the program re-displaying the form with suitable error messages. The redisplayed form input should retain its previous contents. It should also display any errors resulting from the underlying web services. Each error message must have HTML class `error`.

1.2.4 Implementation Restrictions

You must use mustache for rendering all templates. However, you may use any npm modules you find useful; in that case, your submission must be set up so that `npm ci` downloads all the necessary modules.

In this project, all your code must run only on the server; you cannot run any code within the browser.

1.2.5 Look-and-Feel

There are no restrictions on the look-and-feel of the pages.

1.3 Working Project

A working version of the project is available at *<http://zdu.binghamton.edu:2346>* (note that this URL will work only from within the CS network).

[It does have a few rough edges which may be fixed (for example, handling of empty results is not always intuitive for the end user). Please report any problems which seem serious].

1.4 Provided Files

The prj4-sol directory contains a start for your project. It contains the following files:

sensors.js This file will constitute the guts of your project. It should export a function which starts a web server on the port specified by its first argument and use the web services wrapped by the wrapper object specified by its second argument.

sensors-ws.js This provides a wrapper object which wraps the *Project 3* web services. The provided index.js creates an instance of this wrapper object and injects it into your sensors server. You should not need to change this file.

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.

After validating the command line arguments, it instantiates an instance of the web service wrapper using the web service URL provided on the command line. It then calls the function exported by sensors.js passing it the port specified on the command-line and the instance of the web service wrapper.

widget-view.js This file provides a function which takes a JavaScript object and translates it into a view object which can then be rendered using the widget.ms mustance template. Its usage is described by comments within the file and built-in tests.

widget.ms A mustache template to render a view object created by widget-view.js.

mustache.js Provides a wrapper object which renders a view using a mustache template. Usage is documented in the file.

style.css A CSS stylesheet to provide styling to the project pages. Most of the style rules use only HTML element selectors but a few of them use classes. The use of these classes should be clear by examining the source of the pages rendered by the *sample solution* (CS-network only).

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.

1.5 Hints

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

1. You will need a solution to *Project 3* to provide the underlying web services. You may choose to use your own solution or the provided solution. Note that the latter has some enhancements documented in its README to facilitate this project.

2. Once you have decided which solution to use, you will need to decide on the server on which you want to run the web services. If using the provided solution, you may instead choose to use an already running instance at *<http://zdu.binghamton.edu:2345>*, Note that this can only be accessed from within the CS network and will be set up to purge all added data every hour on the hour.

   Once you have decided where you are running the web services, you will need to provide the URL for those web services when starting your program. For example, if running your program on port 2346 and using the provided web services instance, you would start your program using:

   ```
   $ ./index.js 2346 http://zdu.binghamton.edu:2345
   ```

3. Read the project requirements thoroughly. Look at the *sample web site* (CS network only). Use the *view source* control on your browser to reverse engineer the sample web site and understand how the requirements were implemented.

4. The only URL specified for your project is / for the home page. You should come up with suitable URLs for the other pages, or you can simply use the URLs from the sample solution. Think about the URLs and methods used for form submission and how both successful and erroneous form submissions will be handled.

5. Understand how the users-ss application described in class used data to drive the code. Consider a similar approach for this project. In particular, think about suitable data for both sensor-types and sensors which are easy to map into the requirements for widget-view.js while also providing sufficient information for validation.

6. 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/prj4/prj4-sol .
```

7. Change into the newly created `prj4-sol` directory and initialize your project by running `npm init`. Specifically:

```
$ cd prj4-sol
$ npm init -y
```

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

8. Install necessary external packages `cors`, `express body-parser`, `mustache` and `axios`:

```
$ npm install cors express body-parser mustache axios
```

   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.

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

```
./index.js
usage: index.js PORT SENSORS_WS_BASE_URL
```

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

11. Commit your project to github:

```
$ git add .
$ git commit -a -m 'started prj4'
```

12. Work off the users-ss example covered in class to create an `express app`. The app should squirrel away its initial arguments, as well as an instance of the mustache wrapper provided by mustache.js in `app` storage. It should listen on the specified `port` and print out a message with the `listen` handler.

13. Start adding routing to your express `app`. Again, using the users-ss application as an example, set up a directory from where express can serve static files. Create an index.html page in that directory for the home page. Use a browser to verify that you can display the home page at `/`.

    The next few steps describe handling for sensor types. While working on this, keep in mind that the handling for sensors is very similar and look for opportunities to parameterize your work so that it can also be used for handling sensors.

14. Start a handler for the sensor types search page and set up a suitable route. Create a suitable mustache template simply containing something like a title. Have the handler render the template. Verify that you can navigate to this page from your home page.

15. Create a suitable footer template and include it into your page template as a mustache partial. Note that the provided stylesheet assumes that the footer links are set up as a HTML list within a `footer` section.

16. Create a `form` container within your template.

17. Come up with static data which describes the sensor type fields. This data should be suitable for widget-view.js or should be easily transformed to be suitable.

    Start adding the sensor type form widgets to your `form` container. The template should merely iterate over widgets it receives in its view-model and defer the actual rendering of a widget to the `widget` partial.

    Iterate this step until all fields are being rendered correctly.

18. Set things up so that form submission redisplays the page with all form values retained. This should happen pretty much automatically if you use the query parameters object to populate the form values.

19. Add code to validate the values submitted by the form. Set up the validation to be driven using the static data you set up for sensor types. If an error message is related to a specific form widget, ensure that it is displayed next to that widget by adding it to the `opts` options argument to `widgetView()`.

20. Each time the form is displayed, fire off a request to the corresponding web service. Use any debugging technique to verify that the call is being made correctly.

21. Add a section to your template to display the results returned by the web services. Note that the provided stylesheet is set up to display these results in a table using selector `table.summary`.

22. Add scrolling capabilities for the previous and next set of results. This is as simple as using relative links which contain only the query parameters of the links returned by the web services. Note that the provided stylesheet assumes that the scrolling controls have a `.scroll` ancestor.

    Verify operation of the sensor type search functionality.

23. Set up a page for the create / update functionality for sensor types. Notice that the form is very similar to that for sensor types search. Try to reuse the code and mustache template from the search.

24. Set up a handler for the create / update. It is easiest to use the same handler for both displaying the form and submitting the form.

    To facilitate getting the value of the limits fields as an object, set up your bodyparser middleware appropriately:

    ```
    bodyParser.urlencoded({extended: true})
    ```

    By specifying `extended` as **true**, you ensure that the middleware will deliver the values of widgets with names like `limits[min]` and `limits[max]` in a widget named `limits` with `min` and `max` properties.

    The handler should check (`req.method`) whether the form is being submitted or merely displayed. If the latter, it should simply render the view. If the former, it should validate. If the validation fails, it should redisplay with errors. If the validation succeeds it should call the underlying web service and redirect to the search page so that the search displays data for the newly created or updated sensor type. This is an application of the Post/Redirect/Get pattern.

25. Repeat the above for sensor information. Note that you should be able to reuse much of your code for sensor types. Ideally, this reuse should be via parameterization and not cut-and-paste.

26. 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.