1 Project 3

Due: Apr 5, by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

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 familiarize you with implementing a REST-based web service.
- To expose you to the express.js 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 a web server using `./index.js`.

```
$ index.js PORT MONGO_DB_URL [DATA_DIR]
```

where

`PORT` Specifies the port number at which the server should listen for incoming HTTP requests.

`MONGO_DB_URL` A url to the mongo db containing the blog data.

`DATA_DIR` This optional parameter should specify a path to a directory which contains data to be loaded into the db. If specified, all the data in the db is replaced by the data from `DATA_DIR`.

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 blog544-ws.js source file to have it implement the following behavior:

- In order to implement HATEOAS, all non-empty results should have a `links` property which is a list of link objects. Each link object should have the following properties:

  `href` An absolute URL for the linked resource.

`name` A name describing the contents of the linked resource.

`rel` The relationship between this resource and the linked resource. This should be one of the *IANA Link Relations*.

Minimally, the `links` property for a resource should contain a self-link with `rel` and `name` set to `self` and the `href` giving the URL of the resource.

- A `GET` request for the URL `/` should return a JSON object containing a `links` property having the following links:
  - A self-link as described above.
  - A meta-info link with `rel = describedby` and `name = meta`. The `href` property should give the absolute URL for meta-information about the blog service. In the subsequent description, we refer to this `href` as the **meta-info URL**.
  - A category link with `rel = collection` and `name = users|articles|comments`. The `href` property should give the absolute URL for the resource corresponding to the `users|articles|comments` category. In the subsequent description, we refer to these `href`'s as **category URLs**.
- A `GET` request to the meta-info URL should return a JSON object corresponding to the meta-information for the blog enhanced with a `links` property containing a self-link.
- A `GET` request to a category URL should return a JSON object containing a property for the specific category; i.e. it should contain a `users`, `articles` or `comments` property as appropriate. The value of this property should be a list of category objects which satisfy the query parameters of the `GET` request.

  Each category object should be enhanced with a `links` property containing a self-link with `href` set to a URL which can be used to retrieve that individual object. This URL is referred to subsequently as an **object URL**.

  The top-level result should also contain a `links` property, minimally containing a self-link.

  To facilitate paging through results:
  - If further results are possible, the `links` property should contain a next-link with `name = next` and `rel = next` with `href` set to the URL which can be used to fetch the next batch of results.
  - If further results are possible, the top-level result should also contain a `next` property giving the index for the first item in the next batch of results.

    - If previous results are possible, the `links` property should contain a previous-link with `name = prev` and `rel = prev` with `href` set to the URL which can be used to fetch the previous batch of results.
    - If previous results are possible, the top-level result should also contain a `prev` property giving the index for the first item in the previous batch of results.

- A `GET` request to an object URL should return the corresponding object enhanced with a `links` property containing a self-link.
- A `POST` request to a category URL should create an object described by the JSON request body. A successful request should result in a `201 CREATED` status, with the `Location` header set to the absolute URL of the newly created resource and response body set to an empty JSON object.
- A `DELETE` request an object URL should delete the object specified by the URL. A successful request should return a response body set to an empty JSON object.
- A `PATCH` request to an object URL should update the object specified by the URL with the values in the JSON request body. A successful request should return a response body set to an empty JSON object.

All errors should be caught and reported with an appropriate HTTP status with response body containing suitable `status`, `code` and `message` properties.

The behavior of the program is illustrated in this *annotated log*. A working version of the project is available at *<http://zdu.binghamton.edu:2345>* (as usual, this can only be reached from within the CS network).

1.3 Provided Files

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

blog544-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, method definitions or even auxiliary files as required.

index.js This file provides the complete command-line behavior which is required by your program. It requires blog544.js and blog544-ws.js.

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.

1.4 Hints

Feel free to use any tools which you may find useful for this project. Some suggestions:

- Chrome's debugger as documented in earlier projects.
- Restlet chrome client.
- *Chrome JSON Formatter*.

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 users-ws example.

2. The project requirements specifies only the top-level URL `/`. The choice of all other URLs is entirely up to you; decide on what those URLs should be. You should use a regular structure for these URLs, as otherwise your routing will become extremely tedious.

   Your design should be set up to share a handler between multiple categories. So instead of having $2 + 5 \times 3 = 17$ handlers, you can get by with the following 7 handlers:

   (a) A `GET` handler for `/`.

   (b) A `GET` handler for meta-information.

   (c) A `GET` handler for paging through objects of a particular category.

   (d) A `GET` handler for returning a specific blog object in a particular category.

   (e) A `POST` handler for creating a new object in a particular category.

   (f) A `DELETE` handler for deleting a specific blog object in a particular category.

   (g) A `PATCH` handler for updating a specific blog object in a particular category.

3. Start your project by creating a `submit/prj3-sol` directory in a new `prj3-sol` branch of your `i444` or `i544` directory corresponding to your github repository. Change into the newly created `prj3-sol` directory and initialize your project by running `npm init -y`.

```
$ cd ~/i?44
$ git checkout -b prj3-sol  #create new branch
$ mkdir -p submit/prj3-sol  #create new dir
$ cd submit/prj3-sol        #enter project dir
$ npm init -y                    #initialize project
```

This will create a `package.json` file; this file will be committed to your repository in the next step.

4. Use your editor to add a top-level `"type": "module"` entry to the generated `package.json`. Be careful with your syntax as JSON syntax is quite brittle; in particular, watch your commas.

5. Commit into git:

```
$ git add package.json #add package.json to git staging area
$ git commit -m 'started prj3' #commit locally
$ git push -u origin prj3-sol #push branch with changes
                              #to github
```

6. Install necessary dependencies using

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

This 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 already 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.

7. Commit your changes:

```
$ git add package-lock.json
$ git commit -a -m 'added package-lock'
$ git push
```

8. Copy the provided files into your project directory:

```
$ cp -pr $HOME/cs544/projects/prj3/prj3-sol/* .
$ cp -pr $HOME/cs544/projects/prj3/prj3-sol/.* .
```

This should copy in the `README` template, the `index.js`, the `blog544-¬ws.js` skeleton file and a `.gitignore` file into your project directory.

9. Decide whether you want to use your solution to *Project 2* or the *provided solution*:

   - If you are using your solution, copy over all the necessary files. Please note that the provided files assume the existence of `blog544.js`, `blog-error.js` and `meta.js` files.

- If you are using the provided solution, copy over the necessary files:

```
        for f in blog544.js blog-error.js data.js \
                 meta.js validator.js ; \
        do
          cp ~/cs544/projects/prj2-sol/$f . ; \
        done
```

10. You should now be able to get a usage message and start the server:

    ```
    ./index.js
    usage: index.js PORT MONGO_DB_URL [DATA_DIR]
    $ ./index.js 2345 mongodb://localhost:27017/blog544
    listening on port 2345
    ```

    The server will be listening for incoming requests; you can stop it by using ^C.

    [See the provided log for how to start the server in the background while recording its PID in a `.pid` file which can be used to kill it subsequently.]

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

12. Commit your project to github:

    ```
    $ git add .
    $ git commit -a -m 'set up prj3 files'
    $ git push
    ```

13. Open the copy of the skeleton `blog544-ws.js` file in your project directory. It contains code to start the web server, the start of a routing function, error handling code and a utility function. It does not contain code for any handlers.

14. Set up all your routes using the method of the *expressjs app* corresponding to the HTTP method. Note that the same handler can be used for multiple categories as long as the category is passed into the handler. Instead of hardcoding your URLs for the different categories, you can build them dynamically by looping through the `meta` information.

15. Create dummy handlers for all the handlers you specified when setting up your routes.

16. Start out ignoring all link requirements. Complete your meta-info handler ignoring the self link requirement. Simply grab the meta information from the application context and write it into the response object using res.json(). Test and verify.

17. Complete your `GET` handler for object URLs. You should be able get the object id from your object URL using req.params. Then simply `await` the

`find()` method of the blog data store passing in the id. Stuff the returned object into the response as JSON.

18. Create a utility function which when given a request, creates a self-link. Use this utility function to add a self-link to the top-level response in the previous step as well as to the returned object.

19. Complete your handler for `/`. Use the above utility function for the self-link. Build the category links by looping through meta properties.

20. Complete your `DELETE` handler. You should be get the object id from your object URL and then `await` the `remove()` method of the blog data store passing in the id.

21. Complete your `PATCH` handler. You should be get the object id from your object URL, combine it with the request body obtained using `req.body` and then `await` the `update()` method of the blog data store passing in the id.

22. Complete your `POST` handler. You can get the object being created from the request body using `req.body` and await the id returned by the `create()` method of the blog data store. Create an object URL for the newly created object (using the returned id and the URL of the request) and use it to set the `Location` header using res.append().

23. Complete your handler to list all the objects when given a `GET` request to a category URL. You can use req.query to obtain all the query parameters which you can provide to the `find()` method of the blog data store for filtering the results. Add the next and previous links and values. Note that nodejs's querystring.stringify() can be used convert an object to a query-parameters string (without the leading `?`), Test to ensure that you are respecting the `_index` and `_count` query parameters.

24. 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 *git setup* document to submit your project to the TA via github.