1 Project 1

Due: Feb 17 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 ensure that you have set up your VM as specified in the *VM Setup* and *Git Setup* documents.
- To get you to write a 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 blog data stored entirely in memory. The data is organized in three categories:

Users The people associated with the blog. Each person can have one-or-more roles of `admin`, `author` or `commenter` (the `admin` role is not used in this project). Each user is identified using an externally generated `id`.

Articles The blog articles. Will have title and content fields, an authorId field identifying the author and a list of keywords. Each article is identified using an internally generated `id`.

Comments Comments associated with each blog article. A comment will have associated `content`. A comment will be associated with its corresponding article using its `articleId` field and its commenter using its `commenterId` field. Each article is identified using an internally generated `id`.

All data item in each category have `creationTime` and `updateTime` timestamps.

It is possible to perform the following actions on the blog:

Create Create a data item in a specific category.

Find Find zero-or-more data items in a specific category.

Remove Remove a data item in a specific category.

Update Update a data item in a specific category.

Clear Clear all data items in all categories.

All of the actions (except for `clear`) take an object as an argument specifying the details of the action.

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

1.3 Requirements

You must push a `submit/prj1-sol` directory to your `master` branch in 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 blog544.js source file as per the requirements in that file and the requirements in meta.js

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

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 ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

1.4 Command Line Driver

You are being provided with a command-line driver for your code in index.js. This section documents its use.

If you invoke it without any arguments, you will receive a usage message (in this and subsequent logs, the nodejs warning about the experimental ESM loader is not shown):

```
$ ./index.js
index.js [-n|-c|-v] DATA_DIR
$
```

The required `DATA_DIR` is a directory which contains JSON data files for loading into the blog. Specifically, it loads all JSON files with paths matching the following shell wildcard pattern:

`DATA_DIR/json/{articles,comments,users}/*.json`

The options have the following effect:

`-n` **noLoad** If specified, then data is not loaded from `DATA_DIR`. Handled entirely by `index.js`.

`-c` **clear** All data is cleared as the program starts up. Handled entirely by `index.js`. Does not have any effect in this project.

`-v` **verbose** Print details of the use of secondary indexes. Specifically, whenever a secondary index is used a line should be printed containing the name of the indexing field, followed by a `:` and single space followed by a count of the number of objects which have the search value for that field.

This option must be handled by the code you need to implement in `blog544.js`.

When the program starts up, it first prints out a help message and starts up a REPL. The help message summarizes the commands which may be typed into the REPL:

```
$ ./index.js ~/cs544/data
create users|articles|comments NAME=VALUE...
clear clear all blog data
find   users|articles|comments NAME=VALUE... NAME...
help  output this message
remove users|articles|comments NAME=VALUE...
update users|articles|comments NAME=VALUE...
>>
```

The `NAME=VALUE` pairs must meet the requirements in `meta.js`. The `VALUE` cannot contain any whitespace, but it may specify a list using comma-separated values surrounded by `[` and `]` (useful for typing in a value for article `keywords`).

The `find` command allows two additional `NAME`s beyond those specified in `meta¬.js`:

`_count` A limit on the maximum number of returned objects. It defaults to 5. This limit will need to be enforced by your implementation of the `find()` method.

Bug: Currently, there is no validation for this value.

`_json` Specifies a path to a JSON file which is merged into the name/value pairs. If no file is found at that path, then the specified path is appended to `DATA_DIR` and the file is looked for at that path instead.

The `_json` name/value pair is handled completely by `index.js` and will never be seen by your code.

Examples of the command-line interaction can be found in the *annotated LOG*.

1.5 Provided Files

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

blog544.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 blog544.js. You **must not** modify this file; this ensures that your `Blog544` class meets its specifications and facilitates automated testing by testing only the `Blog544` API.

meta.js Meta-information about the different blog object categories. You should not need to modify this file.

validator.js A validator for blog objects. This validator handles all validations which are dependent solely on that object and do not depend on any other objects. You should not need to modify this file.

blog-error.js A simple class which encapsulates an error having a `code` and `message`. You should not need to modify this file.

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 blog data files.

1.6 Hints

You should feel free to use any of the functions from the standard library; in particular, functions provided by the Array, String, Set and Math objects may prove 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 *VM Setup* and *Git Setup* 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. Specifically, use the `--inspect-brk` node option when starting your program and then visit `about://inspect` in your chrome browser.

     There seems to be some problems getting all necessary files loaded in to the chrome debugger. This may be due to the use of ES6 modules. The provided `blog544.js` file has a commented out `debugger` line. If you have problems, uncomment that line and when your program starts up under the debugger use the *return from current function* control until the necessary source files are available in the debugger at which point you can insert necessary breakpoints.

4. Consider what kind of indexing structure you will need to track the blog category. For each category, you will need a collection indexed by the `id` field. Additionally, you will need to set up the indexes required by meta.js for each category,

   When designing your indexing structures, 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.

A Set of ID's may be useful for implementing secondary indexes.

5. Start your project by creating a `submit/prj1-sol` directory in a new `prj1-sol` branch of your `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`.

   ```
   $ cd ~/i?44
   $ git checkout -b prj1-sol  #create new branch
   $ mkdir -p submit/prj1-sol  #create new dir
   $ cd submit/prj1-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.

6. Commit into git:

   ```
   $ git add package.json  #add package.json to git staging area
   $ git commit -m 'started prj1'  #commit locally
   $ git push -u origin prj1-sol  #push branch with changes
                                  #to github
   ```

7. 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.

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

   Commit your changes:

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

8. 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 ~/projects
   $ 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 a file `vm.png`. The captured image should show the terminal window containing the output of the above commands as part of the `x2go` client window. Move the `vm.png` file from your workstation to your VM's `~/?44/submit/prj1-sol` directory (you can use `scp`; if your workstation cannot connect directly to your VM, then do a 2 step copy using `remote.cs` as an intermediate).

Add, commit and push the `vm.png` file.

9. 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` command-line driver, the `blog544.js` skeleton file and the auxiliary files `blog-error.js`, `meta¬.js` and `validator.js` into your project directory.

Also, use

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

to copy *hidden* files, in particular `.gitignore`.

10. 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
$ ./index.js ~/cs544/dat
(node:9810) ExperimentalWarning: The ESM...
create users|articles|comments NAME=VALUE...
clear clear all blog data
find   users|articles|comments NAME=VALUE... NAME...
help  output this message
remove users|articles|comments NAME=VALUE...
update users|articles|comments NAME=VALUE...
>>
```

In some cases, you will also get validation errors: for example

```
>> create users
MISSING_FIELD: missing user ID, user email,
user first name, user last name, user roles fields
for users create
>>
```

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

12. Commit your project to github:

    ```
    $ git add .
    $ git commit -a -m 'added prj1 files'
    ```

13. Open the copy of the `blog544.js` file in your project directory. It contains

    (a) Imports of auxiliary files.

    (b) A comment describing the data model and error codes.

    (c) The code starts out with an `export` directive. This makes `Blog544` the only symbol exported from this file.

    (d) A skeleton `Blog544 class` definition with `@TODO` comments. The comments and `meta.js` 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 `data` in the `constructor¬ ()` we may have a statement like:

      ```
      this.data = {};
      ```

    - 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.

    (e) The class skeleton contains the 7 functions/methods you need to implement. These methods are:

    `constructor()` The constructor for the class. JavaScript requires the name. You may change the parameter list. Note that it creates a validator.

    `make()` A static factory method which invokes the constructor. This is how `Blog544` instances will be constructed externally.

`clear()` A simple method which should clear out all the data stored within the invoking instance of the `Blog544 class`.

`create()`, `find()`, `update()`, `remove()` The main methods you need to implement.

Note that the provided skeletons for these functions already contains partial validation code. This code checks for required/forbidden values, converts strings to `Date`'s as necessary, inserts default values. The validation is driven by the table in the `meta.js` file which was provided as an argument to the `make()` factory function. Note that though this generic validation takes care of most of your validation needs, you will still need some additional validation. In particular, you will need to handle the situations documented at the start of the file for `BAD_ID` and `EXISTS`.

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 `BlogError` objects (for example, see the code at the end of the `validate()` method of the Validator `class`). If you need to `throw` your own errors, be sure to wrap them in a list. Do so even for a single error:

```
        if (this.find(category, { id: obj.id}).length > 0)
{
          const msg =
            `object with id ${obj.id} already exists for
${category}`;
          throw [ new BlogError('EXISTS', msg) ];
        }
```

  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 `*Specs` object which is the argument to the methods in `Blog544` **can** be `String`'s, even when the corresponding property is expected to be a `Date`. For example, a `creationTime` may come in as:

```
        { ...
          creationTime: "2019-08-22T12:02Z",
          ...
          }
```

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

This possibility is handled correctly by the provided validation code with the validation code return value containing the property as a `Date`.

- In contrast to the previous point, any date-valued properties in the return values of the `Blog544` methods should be JavaScript `Date`'s.

14. Add initialization code to your `make()` factory function and `constructor()`. This will set up the necessary indexing structures to handle the requirements for this class.

    There are two approaches you can take to building up your indexing structures:

    (a) A data-driven approach driven entirely by the `meta` information provided to your class. This is the approach taken by the validation code in the Validator `class`.

    (b) A hard-coded approach. You can regard the `meta` information simply as documentation and hard-code the indexing structures into your code.

    To start with, it may be a good idea to merely index by `id` and not worry about the secondary indexes required by meta.js. Specifically for each category, you will need to maintain mappings between ID's and their corresponding objects.

15. Add code for the `create()` method. If you are ignoring secondary indexes, you will merely need to add the validated object to your primary index for that category.

    The `create()` method is responsible for generating IDs for the `articles` and `comments` category. This can be relegated to a utility method. The requirements for the generated id include:

    - It obviously needs to be unique for a specified category.
    - Given a specific ID, other ID's should not be easily guessable. That precludes using a simple sequence of numbers.

    If you decide to use JavaScript's builtin Math.random(), you will receive different ID's each time you run your program. You can avoid this by using an undocumented nodejs option `--random-seed` to provide an initial seed for the random number generator using something like `--random-seed=1`.

    Utility functions like generating ID's do not need access to an instance of `Blog544`. One alternative is that they could be defined as "private" `static` methods. A better alternative is to define them as functions outside the `class` with the advantage of true privacy and not needing to precede each invocation by the class name.

You should now be able to test your `create()` with the command-line program which creates instances of blog objects from the data directory.

16. Implement your `find()` method. For now, simply ensure that you can `find()` by ID. Test.
17. Add code to your `create()` method to use your `find()` method to check for `EXISTS` errors. Test.
18. Add code to your `create()` method to verify that the objects in other categories referenced by a field's `identifies` property really do exist. Test.
19. Implement your `update()` method. Test.
20. Implement your `remove()` method. Be sure to use your `find()` method to validate that the object about to be removed is not referenced by objects of other categories. Test,
21. Add secondary indexes. If the `verbose` option has been specified, print out a line containing the name of secondary index field and number of matching objects when ever the secondary index is used.

    Certain checks are necessary when dealing with secondary indexes:

    - When you remove an object, it needs to be removed from all indexes referring to it.
    - When you update an object, you need to ensure that the object is removed from any secondary indexes for its old values and added to the secondary indexes for its new values. For example, if you update a user `email` field, then you need to ensure that the user is removed from the index for the previous value of the `email` and added into the index for the new value of the `email`.
22. 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.