1 Project 2

Due: July 10 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 give you more experience with JavaScript programming.
- To expose you to mongodb.
- To make you comfortable using nodejs packages and module system.

1.2 Overview

In this project, you will implement two kinds of persistent objects:

A book catalog containing a collection of books. Each book is identified by an ISBN and has additional fields like title, one-or-more authors, publisher, year published, and number of pages. For simplicity, the catalog does not track quantities or allow deletion of items from the catalog.

A shopping cart containing zero-or-more catalog items. It maps a catalog identifier (referred to in industry parlance as a *Stock Keeping Unit* or `sku`) to the quantity of that catalog item in the cart. The cart in this project will only contain books. For simplicity, the shopping cart does not have any prices.

1.3 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.mjs`.

You are being provided with an `index.mjs` which provides the required command-line behavior. What you specifically need to do is add code to the provided model.mjs source file as per the requirements in that file.

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

1.4 Provided Files

The prj2-sol directory contains a start for your project. It contains the following files:

model.mjs 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.

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

index.mjs This file provides the complete command-line behavior which is required by your program. It requires model.mjs. You **must not** modify this file; this ensures that your `Model` class meets its specifications and facilitates automated testing by testing only the `Model` API.

meta.mjs Meta-information about the different model object categories. You should try to avoid modifying this file.

model-error.mjs A trival class for application errors.

validator.mjs Validation code for checking for local errors which depend only on a single object instance. Note that it provides generic validation based on types for input parameters. More validation will be necessary, for checking for global errors across objects.

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 data files which can be used to test your project.

1.5 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 v3.6.8
connecting to: mongodb://127.0.0.1:27017
...
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.
- Given a collection, you can asynchronously find() a cursor which meets the criteria specified by a filter 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.

  If the value of the filter field is an object containing properties for one of mongodb's *query selectors*, then the filter can do more than merely match the find parameters.
- 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.

- Mongo db indexes can be used to facilitate search. In particular, it supports a **single** *text index* on each collection.

1.6 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 users-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. 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. If you do not see all your source files when the debugger starts up, repeatedly use the *return from current function* control repeatedly until the necessary source files are available in the debugger at which point you can insert necessary breakpoints (the critical file will probably be `model.mjs` where you will be doing most of your debugging).

      The provided `model.mjs` file has a commented out `debugger` line. If the above procedure does not work, then uncomment that line and try again.

      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.

    A common cause for development errors is missing a use of `await` before an asynchronous call. Whenever you get an error message about some method not supported by an object, that object may unexpectedly be a `Promise` because you did not use `await`.

3. Consider how you can use mongo to implement this project and use its indexing facilities to access your model data easily.

    Try to use mongo's facilities as much as possible; for example, use mongo's `_id` field to hold object ID's; 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 sort(), skip() and limit() to sort your results and 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 it for the duration of the program. It is also important to remember to close the connection before termination of the program.

[Note that except for the `load` command, the provided command-line program for this project performs only a single command for each program run. This is not typical and will not be the case in future projects. Hence the API provided for `Model` allows for multiple operations for each instance and you should associate the database connection with the instance of `Model`.]

4. Start your project by creating a `submit/prj2-sol` directory in a new `prj2-sol` branch of your `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`.

```
  $ cd ~/i?44
  $ git checkout -b prj2-sol  #create new branch
 $ cp -pr ~/cs544/projects/prj1/prj2-sol . #copy provided files
  $ cd prj2-sol               #go into 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.

5. Commit into git:

```
 $ git add .                           #add directory to git staging area
  $ git commit -m 'started prj2'  #commit locally
  $ git push -u origin prj2-sol   #push branch with changes
                                  #to github
```

6. 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. This should be enforced not only by the `.gitignore` file you copied over when starting the project, but also by the `.gitignore` file at the root of your repository which was set up when you followed the provided directions for 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. 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.mjs
   usage: index.mjs MONGO_DB_URL COMMAND
     where COMMAND is one of
     add-book    NAME=VALUE...
         create or update a book
   ...
   new-cart    NAME=VALUE...
       create a new shopping cart, returning cart id
   ```

   or even do some simple validations:

   ```
       $ ./index.mjs GARBAGE_URL clear
       ***  bad mongo url GARBAGE_URL
       usage: index.mjs MONGO_DB_URL COMMAND
       ...
       $ ./index.mjs mongodb://localhost:27017 add-book \
            isbn=x123 title='some bad book' \
    authors='[author, bad; other author, evil]'
       isbn:BAD_FIELD_VALUE: bad value: "x123":
         The ISBN field must consists of one-or-more digits separated by '-
   '.
      :MISSING_FIELD: missing fields "Publisher", "Publication Year".
       $
   ```

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

10. Commit your project to github:

    ```
    $ git add .
    $ git commit -a -m 'set up prj2 files'
    $ git push
    ```

11. Open the copy of the `model.mjs` file in your project directory. Start by implementing the factory method `make()`. It is set up to call the `Model` constructor with a `props` object which contains properties to be injected into the new `Model` being constructed. As provided, only a single `validator` property is set up; you will need to set up additional properties.

    - Connect to the database to get a client instance.

- Create properties for any collections your design uses.

The final synchronous call of the `constructor()` will cache all the properties you set up in the newly created `Model` instance.

[An instance of a `Model` 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. Setting up indexes is also asynchronous; this too can be done in the `async` factory method].

12. Think of a `try-catch` template for implementing all your action methods. This should be set up to catch any errors and convert them to `ModelError`'s having code `DB`.

13. Implement the `close()` method using a property you have cached within the `Model` instance.

14. Implement the `clear()` method. Since databases and collections come into life automatically in mongodb, you can implement clear by clearing out all of your collections.

15. Implement the `new_cart()` method. All you really need to do is create an ID for the cart being created and store it in the database using something like insertOne(). (do not forget to `await` the asynchronous call).

    The generated ID for the new cart should be a `String` satisfying the following:

    - It must be unique.
    - To avoid security problems it should not be easily guessable.
    - It should not be generated directly by the database as that can make data migration clumsy.

    Uniqueness can be ensured by making an ID contain some kind of sequential counter; to avoid having to explicitly persist the counter, it could be mainained implicitly as the number of entries within some mongo collection.

    Non-guessability can be ensured by including a random portion generated using `Math.random()`.

    Test using the mongo shell to ensure that you are creating carts with the IDs you expect.

16. Implement the `cartItem()` method. You can use mongo's updateOne() methods using mongo's $set and $currentDate operators. You can report a `BAD_ID` error if the `matchedCount` of the update result is different from `1`.

If `nUnits` is 0, you can remove the SKU from the cart. Alternatively, you can simply store the 0 within the cart and filter out the corresponding SKU when returning the cart.

[For now, ignore the requirement for a `BAD_ID` error when the `sku` specifies an unknown ISBN, since you have not yet implemented the book catalog.]

17. Implement the `getCart()` method. You can use mongo's findOne() method. If there is no cart for the specified `cartId`, it will return `null` which you can check to report a `BAD_ID` error.

    If you are storing SKU's having 0 `nUnits`, then filter out those SKU's before returning the retrieved cart.

18. Implement the `addBook()` method. The `upsert` option for mongo's updateOne() is useful to implement the required semantics of insert or update.

    You can test by inserting a single book or by using the `loadBooks` command to load multiple books from a file.

19. Implement the `findBooks()` method. First set up the prerequisite indexing, probably within the `make()` factory method. Once you have done so, mongo's find() method provides the necessary functionality. Pull out any `_index` and `_count` parameters from the `nameValues`, using default values if they are not present. Set up a sort(), skip() and limit() filter on the Cursor returned by the `find()`.

20. Go back to `cartItem()` code and add a check to see whether the `sku` provided is valid using your newly implemented `findBooks()` method to do a search by ISBN.

21. 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 complete, please follow the procedure given in the *git setup* document to merge your `prj1-sol` branch into your `master` branch and submit your project to the grader via github.