1 Project 2

Due: Fri, Apr 2 before midnight.

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.

1.2 Overview

In this project, you will:

1. Make course grades persistent by persisting them to mongodb.
2. Allow specifying multiple courses in different formats specified using a "data" file.

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 db-grades.mjs source file as per the requirements in that file.

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

1.4 Provided Files

You are being provided with a directory prj2-sol directory which contains a start for your project. Additionally, the extras directory contains extra files

germane to the project which need not be submitted. You are also being given two external npm packages courses-info and courses-grades. The data directory has also been updated.

1.4.1 The prj2-sol Directory

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

db-grades.mjs This is the skeleton file which you will need to modify. 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.mjs and cli.mjs These files provide the input-output behavior which is required by your program. It imports db-grades.mjs. You should not need to modify these files.

Note that index.mjs is sym-linked as `interact.mjs`. Running `./interact.mjs` will put you in a REPL similar to that from your previous project. As in that project, it works with grades stored in memory.

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.

1.4.2 The extras Directory

The extras directory contains a *LOG file* which illustrates the command-line behavior of your program.

1.4.3 The courses-info Package

The courses-info package contains "data" about the grading organization for different courses.

- The types used by this package are documented in the typescript course-info.d.ts file.
- Auxiliary functions used by the course descriptions are in the course-info-fns.mjs file.
- Currently available course descriptions include cs544-info.mjs and cs471-info.mjs. Creating additional descriptions should be straight-forward.

The package exports a single `courseInfo()` function which obtains the course information for a specified course.

You do not need to understand this package in detail The provided command-line program cli.mjs obtains a `courseInfo` course information object. Your code need merely pass this `courseInfo` object on to the `course-grades` package described next.

1.4.4 The course-grades Package

The courses-grades package provides a super-set of the solution to *Project 1*. Instead of being capable of only handling cs544, this package can handle any course which has a course description file meeting the requirements given by course-info.d.ts.

In order to use this package in your project, you will need to understand the public methods `make()`, `query()`, `raw()` and `add()` of the `CourseGrades` class in the file course-grades.mjs file.

This package also provides a trivial `AppError` class in a util.mjs file.

The package exports the `CourseGrades` class and the `AppError` class. You will need to import using:

```
import { CourseGrades, AppError } from 'course-grades';
```

1.4.5 Data Directory

The course data directory has been updated. The data files provided for the previous project have been moved to a prj1 directory. Instead, the `*grades*¬.json` file are similar to those for your previous project but do not contain any people information; instead, they contain only the raw grade table for a single course.

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 (as long as mongo's *strict mode* is off).
- 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 get all its results as an array using the asynchronous toArray() method.
- Mongo db indexes can be used to facilitate search.

1.6 Design and Implementation Considerations

The following information may be useful when working on this project.

1.6.1 General Mongo Information

- It is important to emphasize that mongo is promise-ready. Hence if a mongo function is documented to require a callback, then it will usually return a `Promise` when called without a callback which can then be used with `await`.
- Try to use mongo's facilities as much as possible; for example, use mongo's `_id` field to hold object ID's;
- There is no need to explicitly create a database or collection in mongo; both will be created transparently when data is first inserted into a collection in the database. However, any attempt to drop a non-exiting collection will fail with an error. If this is a possibility in your design, then you should check whether a collection exists before attempting to drop it. Note that it is possible to list out the collections in a database by using the synchronous db.listCollections() method followed by an asynchronous `toArray()` on the resulting cursor.
- 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 (this is handled by the CLI program calling your implementation of the `close()` method).

  [Note that 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.]
- As mentioned in class, it is impossible to have an `async` constructor. Hence an instance of `DBGrades` is created using an `async` factory method `make()` which performs all asynchronous operations and then creates the instance using a synchronous call to the constructor. So the factory method will need to perform the following:

  1. Use an asynchronous operation to get a connection to the database.
  2. Call the constructor synchronously passing it the information which the instance will need.

1.6.2 Data Storage Format within mongo

You should only persist raw grades in the database. Review the solution to Question 17 in the *homework 1 solution* for some tradeoffs between different

in-memory representations. A similar analysis can be done when deciding on a format for persisting to mongo:

Single Collection Indexed by Course ID Use a single collection with mongo's `_id` set to the course-id and the associated document containing the raw grades. Advantages and disadvantages:

- Obtaining raw grades for a single course is trivial.
- Updating a single grade in a course involves updating the entire course document. This is not too bad since a course document will typically only be a few KB.
- The representation is biased towards lookup by course.
- Obtaining the grades for a single student across multiple courses will be extremely clumsy.

A Flat Unbiased Collection Use a single collection for all grades across all courses. Each stored document will contain a 4-tuple:

```
{ courseId, studentId, assgnId, grade }
```

In this case, mongo's `_id` will not be used at all. Instead, indexes will be set up for `courseId` and `studentId`. Advantages and disadvantages:

- The representation is not biased in that it allows easy access by either student or course. Specifically, reading all grades for a single course, a single student across multiple courses, or a single student in a single course should be relatively efficient.
- Updating a single grade only changes a single tuple.
- Updating grades entails the hidden cost of mongo updating its indexes.

The first implementation is simplest and will be reasonable for this project and subsequent projects. For a real system the second implementation or other similar alternatives should be evaluated.

1.6.3 Error Handling

All errors should be reported back to the caller by returning an object having an `errors` key with the value of `errors` set to an array of AppError instances.

Note that error handling using this convention is extremely clumsy. The result of each call which may result in an error needs to be checked:

```
const result = someFnWhichMayResultInError();
if (result.errors) {
  //handle errors
}
else {
  //continue with happy path
}
```

Fortunately most of the time the handling of the error can be relegated to the caller and a potential *pyramid of doom* problem avoided using code like:

```
const result = someFnWhichMayResultInError();
if (result.errors) return result;
//continue with happy path
```

You should protect all database operations within a **try-catch** and convert any errors caught within the `catch()` to an `AppError` returned via the `errors` property.

1.7 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 user-store example.

   Also look at an in-memory implementation of your project in the class `InMemoryGrades` given towards the end of the provided cli.mjs.

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 files will probably be `mem-spreadsheet.mjs` or `persistent-spreadsheet.mjs` where you will be doing most of your debugging).

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. Start your project by creating a new `prj2-sol` branch of your `i444` or `i544` directory corresponding to your github repository. Then copy over all the provided files:

```
$ cd ~/i?44
$ git checkout -b prj2-sol #create new branch
$ mkdir -p submit          #ensure you have a submit dir
$ cd submit                #enter project dir
$ cp -r ~/cs544/projects/prj2/prj2-sol .  #copy provided files
$ cd prj2-sol              #change over to new project dir
```

4. Set up this project as an npm package:

```
npm init -y               #create package.json
```

5. Commit into git:

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

[To avoid loosing work, you should get into the habit of periodically `commit`'ting your work and `push`'ing it to github.]

6. Replace the `XXX` entries in the `README` template and commit to github:

```
$ git commit -a -m 'updated README'
$ git push
```

7. Install the mongodb client library using `npm install mongodb@3.6.3`. 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.

[Note that the above installs a specific version of mongo rather than the latest version. Doing so avoids an irritating spurious warning produced by the latest version.]

8. Install the packages provided to support this project:

```
$ npm install ~/cs544/lib/course-grades \
              ~/cs544/lib/courses-info
```

9. You should be able to get a usage message for your project:

```
$ ./index.mjs
index.mjs [--out=text|js|json|json2] DB_URL CMD ...

command can be one of:
   add COURSE_ID EMAIL_ID,COL_ID,VALUE...
   Update COURSE_ID with triples (no space within each triple).
 ...
 raw COURSE_ID
   Return raw grades (no stats) for COURSE_ID.
```

You will also get usage messages for improper individual commands, but a correct command will return with no results because of the unimplemented functionality:

```
$ ./index.mjs mongodb://localhost:27017/grades raw cs544
$
```

Additionally, you should be able to run an interactive in-memory version of your project using `./interact.mjs`:

```
$ ./interact.mjs
>> help

command can be one of:
   add COURSE_ID EMAIL_ID,COL_ID,VALUE...
...
>> raw cs471
BAD_VAL: unknown course "cs471"
>> import cs471 ~/cs544/data/cs471-grades-min-5.json
>> raw cs471
emailId prj1 prj2 prj3 hw1 hw2 hw3 qz1 ...
khurst    88   89   93  89  80  82  10 ...
...
```

```
>>
```

10. Implement the `async` **`make`**`()` factory method. You will need to get a connection to the specified database using mongo's connect() method. Pass the returned value along any other information to your synchronous `DBGrades` constructor.

11. Implement the `close()` method. You should now be able to run your program. If it does not terminate, a possible cause is that your database connection is not being closed.

    If you have handled errors correctly within `make()`, you should be getting reasonable error reports on connection failures:

    ```
    ./index.mjs mongod://localhost:27017/grades raw cs544
    DB: cannot connect to URL "mongod://lo ...
    ```

    Providing a bad port should produce an error message after a timeout.

12. Get some data into your database by first implementing the `import¬ ()` method. Note that the course ID is available via `courseInfo.id`. Consider using the upsert option to simplify importing data for a new course or overwriting the data for a previously imported course.

    Run the command line program to import data. Verify the import using the *mongo shell*.

13. Add functionality to read the data by implementing the `raw()` method. Make sure that you signal a suitable error if an attempt is made to access a course which is not in your database.

14. Implement the `query()` method. You can read the raw grades from your database using the `raw()` method you implemented in the previous step. The actual logic for `query()` can be delegated to an instance of `CourseG- rade` created using the raw data.

    You should be checking for errors after each step which can return errors:

    (a) Reading the raw data.

    (b) Creating an instance of `CourseGrades`

15. Implement the `clear()` method.

16. Implement the `add` method. Again, most of the work can be delegated to an instance of `CourseGrades`. You should use your previously created methods. Do not forget to check for errors after each step.

17. 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 `prj2-sol` branch into your `master` branch and submit your project to the grader via github.