1 Project 2

Due: Oct 7 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. It then provides some background information. 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 asynchronous JavaScript programming, albeit in a cookbook manner.
- To give you some familiarity with using a nosql database.
- To make you comfortable using nodejs packages and module system.

1.2 Requirements

Though the command-line requirements for this project are quite different from your *previous project*, the specifications for the `DocFinder` class you are expected to implement are quite similar.

- Since you are being provided with an implementation of the command-line behavior, this document describes the command-line behavior largely by means of a logfile.
- To prevent DRY violations, the "formal" specifications for the `DocFinder` class are only in the doc-finder.js skeleton file you are being provided with.

Hence this section merely gives an overall view of the expected project behavior, leaving the actual requirements to above doc-finder.js skeleton file.

The main difference from *Project 1* is that your `DocFinder` instances are expected to persist state across instantiations. It should do so using a mongo database. This persistent behavior is illustrated by the usage message for the command line interface:

```
$ ./index.js
usage: index.js DB_URL COMMAND [COMMAND_ARGS...]
  where COMMAND is:
  add-content CONTENT-FILE...
```

```
  add-noise NOISE-FILE...
  clear
  complete SEARCH-TERM...
  find SEARCH-TERM...
  get DOC_NAME
$
```

Note that unlike *Project 1* where all interactions were restricted to a single command-line execution, the interactions in the project will require multiple command-line executions. Hence the program needs to retain state between executions using a database.

The specifications require you to implement methods for each of the above 6 commands, an initialization method, 2 auxiliary methods and a constructor, for a total of 10 externally callable functions.

You must check in a `submit/prj2-sol` directory in your gitlab project such that typing `npm install` within that directory is sufficient to run the project using `./index.js`.

1.3 Provided Files

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

doc-finder.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.

docs-cli.js This file provides the complete command-line behavior which is required by your program. It requires doc-finder.js. You **must not** modify this file.

index.js This is a trivial wrapper which merely calls the above `docs-cli` module. You **must not** 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.

The course data directory containing test data for your project has not changed since the previous project.

1.4 Asynchronous Programming in JavaScript

Asynchronous programming has not been covered in depth as yet in class. Fortunately, because of the recent addition of `async` and `await` to JavaScript, it is possible to do serious asynchronous programming in JavaScript by following two simple rules:

1. An asynchronous function must be declared using the `async` keyword.
2. A call to an asynchronous function must be preceeded by the `await` keyword. The `await` keyword can only occur within a function which has been declared `async`.

A consequence of these two rules is that `async` propagates up the function call chain.

It is worth emphasizing the following points:

- Without exception, all asynchronous functions must be declared `async`; this even applies to anonymous functions.

  Hence if `f()` is asynchronous, the expression `array.map(a => await f(a))` is wrong; it must be replaced by `await array.map(async a => await f(a))`.

- The program will not work if the `await` keyword is omitted when calling an `async` function.

- If the documentation for a nodejs asynchronous library function requires a callback with an initial error argument of the form `(err, value) => ...`, then it cannot be called directly using `await`. Instead, it needs to be promisified as shown below:

```
//grab nodejs's filesystem library.
//fs.readFile not async/await compatible
const fs = require('fs');
//destructure export from util library to grab promisify.
const {promisify} = require('util');

const readFile = promisify(fs.readFile);
//readFile() can now be used with await.
```

- There is no way to call the function which is passed to `forEach()` asynchronously using an `await`. Hence some other looping construct should be used instead.

1.5 MongoDB

MongoDB is a popular nosql database. It allows storage of *collections* of *docu-*

ments to be accessed by a primary key named `_id`. Do not confuse mongodb's use of the term *document* with its different use in this project.

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*.
- Many of the crud operations support an upsert option which makes it possible to combine use of a single operation which implements both insert and update/replace functionality.
- 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.

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.
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.
3. Consider what kind of indexing structure you will need to represent the documents in the database. For each unique non-noise normalized word in all the documents you will need to track the following information:
   - The documents the word occurs in.
   - For each document the word occurs in, track the number occurrences of the word in that document.
   - For each document the word occurs in, track the smallest offset of the word in that document. Note that given the offset of a word occurrence and the original document, you can recover the line for that occurrence by simply looking for the last newline before the offset and the first newline after the offset.

   When designing your indexing structure, you can use MongoDB's key-value collections to support the top-level of your index. Note that you will

probably want to populate the `_id` property yourself with values meaningful to your program, rather than using mongodb's automatically generated values for it.

When given a normalized search term, it should be possible to use the database to directly lookup information for that search term. Getting that information should not involve any kind of search.

Note that the mongodb server will be running in a separate process from your program, possibly on a different host. Hence any database calls will be many times slower than calls local to your program. This performance penalty is obviated somewhat by caching used within the database driver, but it is important to minimize the number of calls to the database when designing your indexing structure. One way to do so is to use the `*Many()` API (like insertMany()) whenever you can. Though not always possible, strive to make the number of database calls for a single `DocFinder` operation independent of the size of the documents involved in that operation.

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 that connection for the duration of the program. It is also important to remember to close the connection before termination of the program.

While performing a single command, it is perfectly reasonable to keep stuff in memory. In particular, it may be a good idea to optimistically read your noise words into memory when the object is first instantiated.

You should use the features of the database to ensure that your noise words are maintained as a set; that is, the database should keep only a single copy of a word even if it is added multiple times.

4. Start your project by creating a `work/prj2-sol` directory. Change into that directory and initialize your project by running `npm init -y`. This will create a `package.json` file; this file should be committed to your repository.

5. 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. 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 gitlab. If you have not done so, please add a line containing simply `node_modules` to a `.gitignore` file at the top-level of your `i480w` or `i580w` gitlab project.

6. Copy the provided files into your project directory:

   ```
   $ cp -p $HOME/cs580w/projects/prj2/prj2-sol/* .
   ```

This should copy in the `README` template, the `index.js`, `docs-cli.js` command-line programs, and the `doc-finder.js` skeleton file into your project directory.

7. You should be able to run the project:

```
$ ./index.js
usage: index.js DB_URL COMMAND [COMMAND_ARGS...]
  where COMMAND is:
  add-content CONTENT-FILE...
  add-noise NOISE-FILE...
  clear
  complete SEARCH-TERM...
  find SEARCH-TERM...
  get DOC_NAME
$ ./index.js mongodb://localhost:27017/docs find betty
time 0 milliseconds
no results
$
```

   As illustrated by the above log, all commands will fail since their corresponding implementations in the `doc-finder.js` skeleton files are `NOP`'s.

8. Start by implementing the `constructor()` for `DocFinder`. Note that contrary to some of the documentation, it is necessary to separate out the database name from the rest of the `dbUrl`. You can do this using a single line destructuring assignment with a capturing regex or `lastIndexOf()`. Remember the separated out values in `this`.

   Note that since the constructor is synchronous, you cannot call the asynchronous connect() method within the constructor. All asynchronous initialization needs to be deferred to the `async init()` method you will work on next.

9. Implement the `init()` method to connect() to the server. Note that with the availability of `async` and `await` you can simply `await` the `client` returned by the `connect()` call. Then use its `db()` method to get a handle for your specific database. Stash these away in `this`.

   You can add code into your `init()` method to also stash away references to the individual collections required by your design. Use the `createCollection()` or `collection()` methods.

10. Implement the `close()` method for `DocFinder`. You simply need to `await` a `close()` on your database `client`.

    Test using the command-line program for a command like `find`. The program should simply exit, after printing failure results. To ensure sanity, it may be a good idea to look at your `client` or `db` objects using a debugger or `console.log()`; you should see large objects having many properties.

11. The `get` command allows the retrieval of a document stored earlier using the `add-content` command. So your design should have some way of storing and retrieving complete documents. This functionality is a good way to get started with mongo's basic CRUD functionality:

    (a) Implement `DocFinder`'s `addContent()` method to merely store the `contentText` in the database ignoring any indexing. Use the `upsert` option to ensure idempotency in a simple manner.

    You can use the *mongo shell* to verify that your document has been added to the database.

    (b) Implement `DocFinder`'s `docContent()` method to retrieve a previously stored document. Implement the specified error handling when the document is not found (note that the `message` can be used with the `Error()` constructor, but the `code` will need to be added as a property.)

    You should now have some useful functionality for the `add-content` and `get` commands.

12. Implement `DocFinder`'s `addNoiseWords()` method. Once again, use the mongo shell to verify its operation.

13. Add code to load all noise words into memory. You can set things up to optimistically load the noise words into memory whenever a `DocFinder` instance is created, or to load them lazily only when needed for an operation and not previously loaded.

14. Implement the `words()` method. As in your previous project, it may be a good idea to implement this as a wrapper around a private `_wordsLow()` method which also tracks word offsets.

15. Based on your indexing design, add indexing functionality to `DocFinder`'s `addContent()` method.

16. Implement `DocFinder`'s `find()` method. Note that it should use your indexing structure to go directly into the database only for those words which occur in its search terms.

17. Implement `DocFinder`'s `complete()` method.

18. 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 *gitlab setup directions* to submit your project using the `submit` directory.