# 1 Project 2

**Due**: Mar 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 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.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.

The API is extremely similar to that for your previous project. The differences include:

- The addition of a `close()` method.
- In addition to the `_count findSpecs` from the previous project, `find()` now also requires support for a `_index findSpecs`. Details are documented within blog544.js
- A change in meta.js to allow `creationTime` as a search field. When it is specified for a `find` command, it is not searched for by equality, but by less than or equal to, Specifically, all returned data would have a `creationTime` older or equal to the specified `creationTime`.

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

## 1.3 Provided Files

The prj2-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, method definitions or even auxiliary files as required.

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

**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 try to avoid modifying this file.

**blog-error.js** A trival class for application errors.

**validator.js** Validation code from the previous project. Note that it provides generic validation based on types for input parameters. More validation will be necessary, especially for validation 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 blog data files. It's content is identical to the previous project.

## 1.4 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 v3.6.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.3
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 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. Specifically, the `$gte` and `$lte` selectors can be used to match values which are `>=` or `<=` the specified value.
- 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.

## 1.5 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. 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 repeatedly until the necessary source files are available in the debugger at which point you can insert necessary breakpoints.

     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 errors which occur while developing your project is missing a use of `await` before an asynchronous call.

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

   Try to use mongo's facilities as much as possible; for example, 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 `Blog544` allows for multiple operations for each instance and you should associate the database connection with the instance of `Blog544`.]

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
$ mkdir -p submit/prj2-sol #create new dir
$ cd submit/prj2-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.

5. Commit into git:

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

6. 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.

7. 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 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.

8. Commit your changes:

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

9. Copy the provided files into your project directory:

```
$ cp -pr $HOME/cs544/projects/prj2/prj2-sol/* .
```

This should copy in the `README` template, the `index.js`, the `blog544.js` skeleton file, the field specifications file `meta.js` and the utility files `blog-error.js` and `validator.js` into your project directory.

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.

    The provided code does have sufficient functionality to get a usage message:

    ```
    ./index.js
    (node:1073) ExperimentalWarning: ...
    usage: index.js MONGO_DB_URL COMMAND
    where COMMAND is one of
      clear     $ ./index.js
      ...
      update users|articles|comments NAME=VALUE...
    ```

    or even do some simple validations:

    ```
    $ ./index.js GARBAGE_URL clear
    (node:1616) ExperimentalWarning: ...
    ***  bad mongo url GARBAGE_URL
    usage: index.js MONGO_DB_URL COMM
    ...
    ```

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

12. Commit your project to github:

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

13. Open the copy of the `blog544.js` file in your project directory. It is pretty much the same as what was provided for your previous project except for the *changes mentioned earlier*. It does contain a definition for `MONGO_CONNECT_OPTIONS` which you should specify when connecting to mongo to avoid warnings.

14. Start by implementing the factory method `make()`.

    - Validate the parameters to the method. The `mongoDbUrl` must be a valid URL of the form `mongodb://HOST:PORT/DB`. If invalid, return a suitable error.
    - If the parameters are valid, connect to the database.
    - Finally, synchronously call the `constructor()`. The constructor should cache the database client connection in the instance and set up instance variables for the database collections you are using.

[An instance of a `Blog544` 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].

You will probably want to use a separate mongo collection for each category of blog data.

15. Since meta.js should not know anything about the database being used, it does not forbid the use of an `_id` field. Modify the constructed validator to do so (ideally, without modifying `meta.js`).

16. Start up your `create()` method. As in your previous project, generate an `id` if one is not provided in the `createSpecs`. Set the `_id` field internal to mongo db to this `id`.

    Use one of mongo's insert*() methods to add the object to mongo. Do not forget to `await` that asynchronous call.

    You could test by using the `load` command. If you use the provided data directory, you may be inserting too many objects to understand any problems you may encounter. It is probably a good idea to test using only a single object which you specify by using the `_json` name-value parameter to specify a path to a file you have set up, or type in the individual parameters on the command-line.

    Once the method works, verify that the object(s) were added using the mongo shell.

17. Implement the `clear()` method. Use the mongo shell to verify that all data has been removed.

18. Implement the `find()` method. For now, ignore indexing; simply passing the `findSpecs` to mongo will automatically achieve most of the filtering requirements for the project.

    Ensure that the returned results do not include mongo's internal `_id` field.

19. Use `mongo`'s cursor modification methods like sort(), skip() and limit() methods to implement the required paging and sorting behavior.

20. Add code to implement the `update()` method using one of mongo's update CRUD operations. You should filter the update using only `_id` and use `$set` to specify the fields being updated.

21. Add code to implement the `remove()` method using one of mongo's delete CRUD operations. You should set things up so that you only remove a single object based on its `_id`.

22. Add validations to ensure that you do not get your database into an inconsistent state:

- Creating an object having the same `id` as an existing object.

  [Note that mongo will catch this by throwing an error having a `code` property set to 11000].

- Creating an object which refers to objects which do not exist.
- Updating an object to refer to objects which do not exist.
- Removing an object referred to by other objects.

23. Modify the `make()` and/or `constructor()` methods to add Mongo db indexes to your collections as specified by meta.js. Note that almost all indexes except `creationTime` will use `+1` as their index spec; `creationTime` will use `-1` to ensure that its index is sorted in non-ascending order.

    The mongo docs say that it is fine to create indexes which already exist (this is the same behavior as for collections). However, I find it useful to have a separate collection which lists the created indexes and use that collection to verify whether indexes already exist before attempting to create them.

24. Modify your `find()` method to use a mongo *query selector* to ensure that searches which specify a value for `creationTime` looks for those objects having `creationTime` less than or equal to the specified value.

25. 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.