1 Project 2

Due: Mar 8 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 `UrlShortener` 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 `UrlShortener` class are only in the url-shortener.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 url-shortener.js skeleton file.

The differences in the requirements for this project and the previous *Project 1* are:

- `UrlShortener` instances are expected to persist across program runs. This should be achieved by using a mongo database. This persistent behavior is illustrated by the usage message for the command line interface:

```
$ ./index.js
index.js SHORTENER_BASE DB_URL COMMAND...
```

```
  where COMMAND is one of:
add          LONG_URL
clear
count        URL
deactivate   URL
query        SHORT_URL
```

- The details of what constitutes a valid URL have been tightened up. Specifically, a URL consists of a **scheme**, **base** and **rest**:

  scheme One or more alphabetic characters terminated by a the sequence `://`.

  base A valid **domain** followed optionally by a `:` and **port**. If present, the **port** must be an integer between 1 and 65535. A valid **domain** is as described in Wikipedia.

  Specifically, a domain consists of one or more labels separated by '.' characters. Each label consists of one-or-more alphanumeric characters `a-z`, `A-Z`, `0-9` or hyphen `-` characters, but not starting or ending with a hyphen. The total length of a domain (including all labels and '.' separators) should be no greater than 253 characters.

- The `remove()` method has been renamed to `deactivate()`.
- There is no requirement that error messages be prefixed with the error code.

The specifications require you to implement methods for each of the above `add`, `clear`, `count`, `deactivate` and `query` commands, a `close()` method and a factory method `make()`, for a total of 7 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:

url-shortener.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 url-shortener.js. You **must not** modify this file.

index.js This is a trivial wrapper which merely calls the above `shortener-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.

1.4 Asynchronous Programming in JavaScript

Asynchronous programming is being covered in class concurrently with this project. 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. Any call to an asynchronous function (one which returns a `promise` or has been declared using the `async` keyword) 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 *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*.

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 urls in the database. You need to be able to look up information for a URL-mapping using both the short and long URL's. For each mapping, you will need to track its query-count and whether or not it is currently active.

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

   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.

   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.

   [Note that the command-line program for this project performs only a single command for each program run. Nevertheless, the API provided for `UrlShortener` allows for multiple operations for each instance; hence you should associate the database connection with the instance of `UrlShortener`.]

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 `i444` or `i544` gitlab project.

6. 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`, `shortener-cli.js` command-line programs, and the `url-shortener.js` skeleton file into your project directory.

7. You should be able to run the project:

```
$ ./index.js
index.js SHORTENER_BASE DB_URL COMMAND...
  where COMMAND is one of:
add          LONG_URL
clear
count        URL
deactivate   URL
query        SHORT_URL
$ ./index.js sadaa asdasdsa add sdadsa
{ code: 'UNIMPLEMENTED', message: 'add() not implemented' }
$
```

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

8. Implement utility methods to encapsulate common functionality like:
   - Verifying a domain.
   - Verifying a port.
   - Splitting a URL into components, checking for errors.
   - Wrap an error code and error message into a error object.
9. Start by implementing the factory method for `UrlShortener`.
   - Validate the parameters to the method. The `mongoDbUrl` must be a valid URL which uses the `mongodb` scheme and `shortenerBase` must be a valid base. If invalid, return an error as per the specs.
   - If the parameters are valid, connect to the database (note that you will need to split `mongoDbUrl` into the base part and the database name).
   - 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 `UrlShortener` should contain a database connection, but obtaining a database connection is an asynchronous operation. Since it is not possible to have an *asynchronous constructor*, an `async` factory method provides a workaround].

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

    Test using the command-line program for a command like `add`. The program should simply exit, after printing out that `add` is not implemented. 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. Implement a method to shorten a long url into a short url.

12. The `add()`, `count()`, `deactivate()` and `query()` methods will all need to look up the database for a URL-mapping. It is a good idea to encapsulate this lookup into a common function. Implement such a function. The details will depend on the indexing structure you have designed for your database.

13. Implement the `add()` method as a wrapper around the utility method from the previous step. You can use the *mongo shell* to verify the contents of your database.

14. Implement the remaining `count()`, `deactivate()` and `query()` and `clear¬ ()` methods.

15. 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.