1 Project 2

Due: Mar 2 by 11:59p

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. Subsequently, it provides a log of the project in operation to help understand the requirements. It then describes the files you have been provided with. 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 familiarize you to asynchronous programming in JavaScript.
- To expose you to using a package manager for JavaScript.
- To provide you with some experience using a popular no-sql database.

1.2 Requirements

This project requires you to implement an interface to a mongo database to allow storing and retrieving images. An image is identified by two strings:

group A non-empty string which does not contain any `NUL` characters.

name A non-empty string which does not contain any `NUL` or `/` characters

An example identification for an image might have *group* `inputs` and *name* `rose`. Since *name* cannot contain a `/` character, we can unambiguously use the single string *group*`/`*name* to identify an image, for example `inputs/rose`.

The image types which must be handled are PNG and PPM. Note that the identification of an image does not include its type; i.e. `inputs/rose` could refer to either a PNG or PPM image.

Set up your gitlab project so that it contains a `prj2/img-store` subdirectory within the top level `submit` directory. It should be possible to install your project by cloning your gitlab project, setting the current directory to its `submit/prj2/img-store` subdirectory and then running `npm install`.

After the install, the `submit/prj2/img-store` should contain a `index.js` executable file which can be run with the sub-commands:

`./index.js get` ***group name type*** Print on standard output the binary *type* contents of the image previously stored in database under id *group*`/`*name*.

`./index.js list` ***group*** Print on standard output the names of all the images stored under group *group*, with each name being output on a separate line.

`./index.js meta` ***group name*** Print on standard output meta-information about the image previously stored under id *group/name*. The output should consist of *key=value* lines for the following keys: `width`, `height`, `maxNColors`, `nHeaderBytes`, `creationTime`.

`./index.js put` ***group imgPath*** Store the image in file *imgPath* under *group* with *name* formed from the basename of *imgPath* with the extension removed.

All data should be stored in the database called `images` at the mongo url `mongodb://localhost:27017`.

The program should detect errors in the syntax of the commands or their operation as documented in the provided file img-store.js.

1.3 Example Log

The behavior of the program is illustrated by the following annotated log:

```
#output usage message
$ ./index.js
usage: index.js get GROUP NAME TYPE
               list GROUP
               meta GROUP NAME
               put GROUP IMG_PATH
#store image inputs/rose
$ ./index.js put inputs ~/cs580w/projects/prj2/aux/images/rose.png
#store another image inputs/logo
$ ./index.js put inputs ~/cs580w/projects/prj2/aux/images/logo.ppm

#list names of all images stored under group inputs
$ ./index.js list inputs
rose
logo
#show meta-info for image inputs/rose
$ ./index.js meta inputs rose
width=70
height=46
maxNColors=255
nHeaderBytes=13
creationTime=2018-02-20T18:27:17.820Z
#show meta-info for image inputs/logo
$ ./index.js meta inputs logo
width=640
height=480
maxNColors=255
nHeaderBytes=15
```

```
creationTime=2018-02-20T18:27:46.055Z
#retrieve ppm version of inputs/logo and store in t.ppm
$ ./index.js get inputs logo ppm >t.ppm
#is identical to original image
$ cmp t.ppm ~/cs580w/projects/prj2/aux/images/logo.ppm
#retrieve ppm version of inputs/rose and store in t.ppm
$ ./index.js get inputs rose ppm >t.ppm
#is identical to original image
$ cmp t.ppm ~/cs580w/projects/prj2/aux/images/rose.ppm
#bad image name
$ ./index.js get inputs logo/ ppm
BAD_NAME: bad image name 'logo/'
#bad image type
$ ./index.js get inputs logo gif
BAD_TYPE: bad image type 'gif'
#bad image type in imgPath
$ ./index.js put inputs ~/cs580w/projects/prj2/aux/images/rose.gif
BAD_TYPE: bad image type 'gif' in path /home/.../rose.gif
#bad image path
$ ./index.js put inputs ~/cs580w/projects/prj2/aux/images/rose1.ppm
NOT_FOUND: file /home/.../rose1.ppm not found
#create a garbage image
$ echo 'hello' > garbage.ppm
#bad image format
$ ./index.js put inputs garbage.ppm
BAD_FORMAT: bad image format
$
```

1.4 Provided Files

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

.gitignore A `.gitignore` file set up to tell `git` to ignore the `node_modules` directory.

img-store.js A skeleton file. You must complete the `TODO` sections in this file so that it exports a `async` factory function. When this factory function is called with no arguments, it should return an object which implements the following `async` methods `close()`, `get()`, `list()`, `meta()` and `put()` with specifications as documented in the file.

img-store-cli.js A file which implements the required command-line interface for the program. You should not need to change this file.

index.js A file which is used as an entry point for the program. It is simply a wrapper for `img-store-cli.js`. You should not need to change this file.

ppm.js This file contains a constructor to build a `Ppm` object which represents an abstraction of a PPM image. It is similar to that which you were provided with in *Project 1*, except for the following changes:

- The constructor must always be called with its first argument set to an ID for the PPM image to be constructed.
- The constructed object does not have a `pixelBytes` property. Instead it has a `bytes` property containing all the bytes of the image (including header bytes). An additional property `nHeaderBytes` gives the number of header bytes contained in `bytes`.
- If there is an error in the format of the image bytes specified by the second argument of the constructor, the constructor returns an error object which meets the specifications of the error objects returned by functions in `img-store.js`.

You should not need to change this file.

README A README file which must be submitted along with the project. It contains a 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 completing the header, you may append any additional content which you would like read during the grading of your project.

The `aux` directory contains an images directory which contains sample images in both PNG and PPM formats.

1.5 Hints

The following points are not prescriptive in that you may choose to ignore them as long as you meet all the project requirements.

The provided files take care of all command-line processing and the reading and parsing of PPM image files. All you need to do is implement the `TODO` placeholders in the img-store.js file.

You may proceed as follows:

1. Study the example which will be covered in class. The example is chosen to cover all the facets of this project.
2. Familiarize yourself with mongodb. It is available for multiple languages; you will need to look at the documentation for the nodejs version. Minimally, you should read the *Quick Start* guide, glance at the tutorials and be ready to refer to the Reference section as necessary. Note that the

mongo server has already been installed on your VM; you will need to install the mongo client locally within your project directory using `npm`.

3. Play briefly with the *mongo shell*. You can start it from the command-line by simply typing `mongo`.

4. Using web resources, get some familiarity with the **Node Package Manager** `npm`. Note that we will merely be using `npm` to install dependencies and not for creating packages.

5. It is recommended that instead of using callbacks or promises, you use `async` and `await`. So familiarize yourself with `async` / `await` and their relationship with promises. Note that if a mongo method requires a callback, it will automatically return a promise if called without a callback. For nodejs use the provided `promisify()` method to convert a function which requires a callback to one which returns a promise.

6. Start your project by copying the prj2 directory to your `work` directory. After doing so, you should be able to run `index.js` and verify that it detects errors in the command-line arguments.

7. Initialize your project with npm. Go into your `work/prj2/img-store` directory and initialize it for `npm` by typing the command `npm init -¬ y`. This will create a `package.json` file. Start installing dependencies; minimally you will need a mongo client which you can install using `npm install --save mongodb`. Commit the generated `package.json` and `package-lock.json` file to your git repository; do **not** add the `node_¬ modules` directory.

8. Start implementing the required factory method. You should have it create a mongo client and db. Store these in the return'd object. If you run your program, it should hang because of open database connections.

9. Implement the `close()` method to close the database connections. When you run your program it should no longer hang.

10. Decide on the collection(s) you want to use in your `images` database. Note that the specifications require that you should be able to implement the `list()` and `meta()` methods without reading image bytes from the database.

11. Start implementing the action methods. Simply use the provided utility methods in `img-store.js` to check for errors in the incoming arguments for each method. Test.

12. Start implementing the `put()` method. To start with, simply work with only PPM images. Use nodejs's `fs.readFile()` method to read the contents of the `imgPath` into a nodejs `Buffer`. Note that `fs.readFile()` requires a callback, but you can use nodejs's `promisify()` method to make it return a promise and thus usable under a `await`. You can verify

that your read worked by having your program temporarily output the length of the return'd buffer.

13. Convert the nodejs `Buffer` to a `Uint8Array` using the `Uint8Array(¬ buffer)` constructor. Use the `Ppm()` constructor to build a `Ppm` object passing it the image id (use utility method `toImageId()`) and the `Uint¬ 8Array` constructed from the image bytes. Verify by having your program temporarily output the results of the `toString()` method on the constructed `Ppm` object.

14. Store the meta-information from the `Ppm` object in the database with `_id` set to the image id. Use the insertOne() method. Use the mongo shell to verify your addition.

15. Attempt to store the same image again. If you have implemented your `put()` correctly, you will receive an exception. You can convert this to a defined error object by verifying that the exception object has a `code` property with value 11000. Any other exception should be rethrown.

16. Decide on the representation of the image bytes in the database. Some possibilities:

    - Have the mongo document contain the image bytes directly.
    - Have the mongo document contain a base64 representation of the image bytes.
    - Have the mongo document contain a `BSON Binary` representation of the image bytes.

    Once you have decided on the representation, use `insertOne()` once again to insert that representation into the database. Again make sure that you catch and convert any database errors.

17. Use mongo's `find()` method to implement the `list()` method. Note that you can convert the results of `find()` to a **promised** array using `toArray()`.

18. Use mongo's `find()` method as above to implement the `meta()` method. Note that in this case, you should expect 0 or 1 results; if 0, then throw a suitable `NOT_FOUND ImgError`.

19. Implement the `get()` method to retrieve the bytes of the image as in the previous step. Convert your database representation of the image bytes to a `Uint8Array`.

20. Implement conversion between image types. First decide on a strategy for conversions. Some possibilities:

    Eager Conversion When an image of a certain type is stored using `put¬ ()`, eagerly convert it to other type(s) and store the bytes for all types in the database.

Lazy Conversion Store only the bytes of the image type explicitly provided to `put()` in the database. Peform conversion lazily only when it is necessary to get the bytes of an image using a type not explicitly stored in the database.

If you go with lazy conversion, then another design decision is whether after a conversion, you cache the converted result in the database or whether you always do the conversion.

To do the actual conversion, use the *Image Magick* convert program to convert between image types by simply having your program run the command `convert` *srcImgPath destImgPath*.

The destination *destImgPath* for the `convert` program should be a temporary file which can be deleted once its bytes have been read into the database. You can use nodejs's `os.tmpdir()` to retrieve the temporary directory where you can create temporary files. Use nodejs's `fs.unlink()` to clean up the temporary file when it is no longer required.

21. Iterate until you meet all requirements. Verify that you detect errors and return appropriate error messages.