Aims

The aims of this project are as follows:

- To verify that you have GUI access to your VM.
- To expose you to programming in JavaScript.
- To encourage you to use advanced features of JavaScript in your code.
- To introduce you to *Test-Driven Development TDD*.
- To familiarize you with the development environment on your VM.

Future Projects

This project requires you to write a program to maintain one or more checking accounts in memory (all data disappears when the program terminates). In future projects, we will be:

1. Storing the accounts in a persistent database so that data survives program termination.
2. Making the accounts accessible via web services.
3. Adding browser UI(s) which use the web services to access the accounts.

Scanned with CamScanner

Requirements

You must push a `submit/prj1-sol` directory to your `main` branch in your github repository such that typing `npm ci` within that directory followed by `./index.mjs` within that directory is sufficient to run your project.

You are being provided with an index.mjs and main.mjs which provide the required input-output behavior. What you specifically need to do is add code to the provided accounts.mjs source file as per the requirements in that file.

Your project should pass all the provided tests.

Your emphasis in this project should be on correctness rather than efficiency.

The behavior of your program is illustrated in an *annotated log*.

Additionally, your `submit/prj1-sol` **must** contain a `vm.png` image file to verify that you have set up your VM correctly. Specifically, the image must show an x2go client window running on your VM. The captured x2go window must show a terminal window containing the output of the following commands:

```
$ hostname; hostname -I
$ ls ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

You will loose 10 points if you do not include the `vm.png`.

Scanned with CamScanner

Provided Files

The prj1-sol directory contains a start for your project. It contains the following files:

accounts.mjs

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.

Test files

This directory contains tests for some of the project functionality using the mocha testing framework and chai assertions. You may add your own tests in this directory.

index.mjs and *main.mjs*

These files provide the input-output behavior which is required by your program. It imports accounts.mjs.

You should not need to modify these files.

util.mjs

Utilities including validation and ID generation functions. You should not need to modify this file, but may find the provided function useful for your project.

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.

Scanned with CamScanner

Course Library

Two packages have been installed in the lib directory:

cs544-js-utils

Utilities which are useful in any JavaScript environment; specifically utilities which are useful within both a browser and node environment. Currently it contains a class `AppErrors` for application errors.

cs544-node-utils

Utilities which are useful in a node environment. Currently it contains code to synchronously read a JSON file.

Scanned with CamScanner

Design and Implementation

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

Use of ES6 Classes

At the time of assigning this project, JavaScript objects and ES6 classes will not have been covered in detail in the course. However, you should be able to use ES6 **`class`** syntax without needing to understand the finer points.

Note that the `class` syntax is a relatively recent addition to JavaScript and is syntactic sugar around JavaScript's more flexible object model. Note that even though the use of this `class` syntax may make students with a background in class-based OOPL's feel more comfortable, there are some differences worth pointing out:

- All "instance variables" must be referenced through the `this` pseudo-variable in both `constructor` and methods. For example, if we want to initialize an instance variable `_accounts` in the `Accounts` `constructor()` we may have a statement like:

```
this._accounts = {};
```

- There is no implicit `this`. If an instance method needs to call another instance method of the same object, the method **must** be called through `this`.
- There is no easy way to get private methods or data in the current language standard (though this should change very shortly). Instead a convention which is often used is to prefix private names with something like a underscore and trust `class` clients to not misuse those names.

Scanned with CamScanner

Pure Functions

Strive to make your functions pure. That is:

- They should not change any non-local state.
- They should not change their arguments.
- They should not change objects unless required by the API.

Note that you can use destructive assignment **within** a function. The purity requirement prevents any destructive effects being caused **outside** the function.

Tests

The provided code contains tests for the `accounts` methods. Most of these tests will initially fail. As you develop your code, you will make the tests succeed, thus practising TDD. You are welcome to add more tests (note that different tests than those provided here may be used when grading your project).

If you set up your `package.json` as instructed, typing `npm test` should run all tests using the <https://mochajs.org/> mocha testing framework.

Note that you can select which tests should run by qualifying `describe` and `it` with `.only`; i.e. changing a `it(...)` or `describe(...)` to `it.only(...)` or `describe.only(...)` selects only those tests enclosed within those function calls to run.

Scanned with CamScanner

Representing Money

Since floating point arithmetic is only approximate, it is usually a bad idea to use floating point arithmetic for financial quantities. For example, adding 0.1 to 0 ten times does not result in a sum of 1:

```
> Array.from({length: 10}).fill(0.1)
    .reduce((acc, a) => acc + a)
0.9999999999999999
> Array.from({length: 10}).fill(0.1)
   .reduce((acc, a) => acc + a) === 1
false
>
```

So money should be represented as integers provided they do not overflow. JavaScript does not have integers, but the `Number` type can represent integers exactly up till 2^{53}.

So we can represent money in "cents". As long as we are merely doing addition and subtraction the integers representable within `Number` should suffice. If that turns out not to be the case, then we can use the `BigInt` class from the standard library.

Scanned with CamScanner

Hints

- You should feel free to use any of the functions from the standard library; in particular, functions provided by the Array and String objects may prove useful. You should not use any nodejs library functions to make it possible to run this code within the browser in future projects. You should also not need to include additional npm packages.
- The requirements in accounts.mjs ask you to implement some methods on an `Accounts` instance representing a collection of `Account`'s, but most of your effort will be . However, most of these methods operate on an individual account identifies by `params.id`.

  Hence it is probably a good idea to define your own `Account` class. The methods implemented within `Accounts` should merely pull up an `Account` instance and delegate all work to methods on the `Account` instance.

he following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. If you have not already done so, set up your course VM as per the instructions specified in the *VM Setup* and *Git Setup* documents.
2. Understand the project requirements thoroughly including the comments in accounts.mjs. Verify your understanding by looking at the provided *interaction log*.

   Also look at the code in main.mjs. You should study the code sufficiently to understand how typing a command on the command-line will result in your code being called. Doing so will also allow you to use similar coding techniques to write your code.
3. 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.

Scanned with CamScanner

- 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.

  Occasionally, there are problems getting all necessary files loaded in to the chrome debugger. When your program starts up under the debugger use the *return from current function* control until the necessary source files are available in the debugger at which point you can insert necessary breakpoints.

It is well worth spending a few minutes setting up the chrome debugger as it could save you a lot of debugging time.

4. Start your project by creating a new `prj1-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 prj1-sol #create new branch
$ mkdir -p submit          #ensure you have a submit dir
$ cd submit                #enter project dir
$ cp -r ~/cs544/projects/prj1/prj1-sol .  #copy provided files
$ cd prj1-sol              #change over to new project dir
```

5. Set up this project as an npm package:

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

6. Commit into git:

```
$ git add .   #add contents of directory to git
$ git commit -m 'started prj1' #commit locally
$ git push -u origin prj1-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.]

Scanned with CamScanner

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

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

8. Capture an image to validate that you have set up your course VM as instructed. Within a terminal window in your `x2go` mainent window, type in the following commands:

```
$ hostname; hostname -I
$ ls ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

Use an image capture program on your **workstation** to capture an image of your `x2go` mainent window into a file `vm.png`. The captured image should show the terminal window containing the output of the above commands as part of the `x2go` mainent window. Move the `vm.png` file from your workstation to your VM's `~/?44/submit/prj1-sol` directory (you can use `scp`; if your workstation cannot connect directly to your VM, then do a 2 step copy using `remote.cs` as an intermediate).

Add, commit and push the `vm.png` file.

9. Update your package.json to allow running automated tests:

   a. Install testing packages:

```
npm install -D mocha chai
```

   - `mocha` is a popular JavaScript testing framework. It requires using an external library for checking whether a test is successful.
   - `chai` is an external library which allows checking whether the result of a test is as expected.

Scanned with CamScanner

- `chai` is an external library which allows checking whether the result of a test is as expected.
- Since testing is a development time activity, the `-D` option is used to install the packages as **development dependencies**. Specifically, they will not be included within the project's **runtime dependencies**.

b. Replace the `script` property in your generated `package.json` with the following:

```
"scripts": {
  "test": "mocha test/",
  "debug-test": "mocha --inspect-brk test/",
  "syntax-check":
    "shopt -s globstar; for f in **/*.mjs; do node -c $f; done;"
},
```

- The "`test`" property specifies the test command. You can run all the tests by simply running the command `npm test`. If you want to run only a particular test or test suite, change the `it` or `describe` to `it.only` or `describe.only` respectively.
- The "`debug-test`" property allows attaching a debugger when testing. You can run this script using the command `npm run debug-test`.
- You may make changes in several files and then find tests are broken because of a syntax error in one of the files. The error messages are not particularly useful in identifying the syntax error. This "`syntax-check`" property provides a script which runs syntax checks for all the `.mjs` files in your project. You can run this script using the command `npm run syntax-check`.

10. Install course library dependencies:

```
$ npm install ~/cs544/lib/cs544-js-utils \
              ~/cs544/lib/cs544-node-utils
```

Scanned with CamScanner

11. You should be able to run the project, but except for error checking performed by `main.mjs`, any input will simply return empty results until you replace the TODO sections with suitable code.

```
$ ./index.mjs
Allowed commands are
   account id=ACCOUNT_ID
...
>>
```

You should also be able to run the provided tests, but most tests will fail.

```
$ npm test
```

12. Open your copy of the `accounts.mjs` file in your project directory. It contains

    a. Imports of auxiliary modules.

    b. A comment documenting the data model.

    c. A default export of a `makeAccounts()` function which is set up to return an instance of the `Accounts` **`class`**.

    d. Skeleton definitions for the `Accounts`, `Account` and `Transaction` classes.

Some points worth making about the provided code:

- The reason for having the exported `makeAccounts()` function is to ensure that the `Accounts` class is not exposed at all to the clients of the accounts module, except via the methods on its instances.
- For error handling, the convention used is that errors are returned by returning an object having an `errors` property containing a list of error objects, each having a `message` property giving a suitable error message and possibly a `options` property. A particularly important option is `code` which returns a code associated with the error.

Scanned with CamScanner

The utils.mjs from the `cs544-js-utils` library package provides an `AppErrors` class which can be used to construct errors objects.

13. Implement the constructor for `Accounts`. You will need to track the contained `Account` objects.

14. Implement the constructor for `Account`. The `genId()` function from util.mjs can be used to generate the ID for a new account. The instance will also need to track its `holderId` and its transactions.

15. Implement the `newAccount()` method in the `Accounts` class. It will need to validate that `params.holderId` is specified; if not it should return a `BAD_REQ` error. If everything is ok, it will need to create a new `Account` instance and return its ID. The `Accounts` instance will also need to track the newly created `Account` instance.

16. Implement the `account()` method in the `Accounts` class. It will need to verify that `params.id` is specified returning a `BAD_REQ` if that is not the case. Otherwise, verify that `params.id` specifies the ID of an existing account tracked by the `Accounts` instance; if not return a `NOT_FOUND` error, else return the existing account instance.

17. Implement the `info()` method in the `Account` class. For now, simply hardcode the returned balance as 0. You should now be able to pass the tests in basic-accounts.mjs.

18. Before dealing with transactions, it may be a good idea to create a `Transaction` class to encapsulate all `Transaction` logic. Internally within the `Transaction` class you can represent the transaction amount in "cents", but you should set it up so that the amount is represented externally as a regular `Number`.

    The `Transaction` instance should also provide a way for comparing two transactions by `date`

19. Implement the `newAct()` method. After validating the parameters, create a new `Transaction` and track it within the `Account` instance. You may want to keep the transactions sorted by date.

    You should now be able to run most of the tests in transactions.mjs.

Scanned with CamScanner

19. Implement the `newAct()` method. After validating the parameters, create a new `Transaction` and track it within the `Account` instance. You may want to keep the transactions sorted by date.

    You should now be able to run most of the tests in transactions.mjs.

20. Implement the account balance which was previously hardcoded to 0. You can do so by simply adding up all the transaction amounts. Note that the addition should be done using cents, but the final balance should be set to a `Number`.

    You should now be able to run all the tests in transactions.mjs.

21. Implement the `query()` method. First validate the provided parameters. If they validate then use the provided parameters to filter the transactions. Finally, return the transactions specified by the `index` and `count` parameters.

    You should now be able to run all the tests in query.mjs.

22. Finally, implement the `statement()` method. This is very similar to the `query()` method but has the additional requirement of computing the account balance after each transaction.

23. Iterate until you meet all requirements and pass all tests.

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.

Scanned with CamScanner