CS 284, Interim 2018, R. Brown

Homework 1 Due Thursday, 1-11-18

Create a directory ~/MCA/hw1 for your work on this homework.

A. ES6 Promises

1. Corrected code for the `Promise` example in class was posted on Piazza.
   a) Enter that code into a file `promise.js` in your `hw1` directory, and run it using your `node.js` installation at the command line as follows:

   ```
   node promise.js
   ```

   Verify that it behaves as expected.
   b) Modify your code `promise.js` by adding more expressions involving the function `timeoutMsg()` and `.then()`, and verify that those expressions also operate as expected.
2. Create a git `commit` of your work on this part, with an appropriate commit message.

B. Git merge requests

1. Carry out the steps for `Inner.java` described in the class notes page for 1/9/18, except for the program `Demo.java`.
   a) Choose changes that seem unlikely to lead to merge conflicts, e.g., choose button names and adapter class names that include your initials or username.
   b) Instead of making a trivial change, implement a more significant change as indicated in the instructions on the notes page.
   c) Recruit others to make comments on your merge request.

C. ES6 fetch() interactions with a server

1. Create a file `fetch-demo.js` with the `fetch` demo code distributed via Piazza
   `https://piazza.com/class/jby2shah6hn6m0?cid=20`
   a) Run this code on your `node` interpreter (make sure `react-install` is a subdirectory of the directory for the following command):

   ```
   node fetch-demo.js
   ```

   This code should print the two one-line messages

   ```
   it worked!
   {count:  2559 }
   ```

   (where `2559` would be replaced by the current count).
   b) Create a git `commit` of your working version of `fetch-demo.js` to record your solution to this part. Include `HW1 C1` in your commit message
2. We will modify `fetch-demo.js` to create a reusable Javascript module. For this exercise, we will use the old `require()` style of module for this exercise instead of the `export/import` style presented in class notes.

a) Edit `fetch-demo.js` as follows:

- Be sure first line, with `require('node-fetch')`, is *not* commented out.
- Instead of using `let` for the functions `getCount()` and `addCount()`, make these functions members of an object `exports` as follows:

```
exports.addCount = () => ...
    ...
exports.getCount = () => ...
    ...
```

- Comment out the two calls at the end `addCount()` and `getCount()`.

b) Now test your module using `node` interactively as follows:

```
node
> let fd = require('./fetch-demo')
> fd.getCount()
> fd.addCount()
> fd.getCount()
> fd.addCount()
```

c) Commit the updated version of your `fetch-demo.js`

3. Besides the fetch example server used for `fetch-demo.js`, a second example server may be found at URL `http://162.210.90.19:3000/names` which offers two types of service request.

- `POST` requests enable a client to submit a string (e.g., their name) to be added to a list maintained by the server.
- `GET` requests return that list of strings.

In the following exercises, we use `fetch` to perform ES5 requests for that server.

a) Add a third function `getNames()` to `fetch-demo.js` that requests the server's list of names.

- Start by copying the definition of `getCount()` in `fetch-demo.js`, and changing the function name in that copy to `getName()`. Use `node` to check that `getName()` works and performs the same as `getCount()`.
- Before making further changes to `getName()`, consider how the original `getCount()` function works.

```
let getCount = () => {
   fetch("http://162.210.90.19:3000")
      .then((res) => {
          return res.json()
      })
      .then((data) => {
          console.log(data)
      })
}
```

  - The `fetch()` call performs a `GET` request on the count server with URL `http://162.210.90.19:3000`.
  - There are two `.then()` calls. The first of these handles the server's response from that `GET` request, by extracting the response such as `{count: 2559 }`. This rsponse string is in *JSON* format, which provides key-value pairs. In this case, a single key-value pair is provided: `count` is the key, and `2559` is the value, indicating that the server's count variable held 2559 at the time of that request.

Note: JSON format looks very much like a Javascript object! However, (1) a JSON expression is always a string in the Javascript language, not an object, and (2) functions can be called in order to compute values for a Javascript object's data members, and such objects can have callable functions as members (i.e., methods).

- The second `.then()` call in this chain prints that JSON value on the `node` console (standard output in our case).

- For `getNames()`, we want to change the name of the server to `http://162.210.90.19:3000/names` in the `fetch()` call. However, it seems likely that the two `.then()` calls will succeed for names as they did for counts, since
  - the server's response will presumaby be available in JSON format via `res.json()` as before, and
  - `console.log()` can probably print that JSON response.
- So, change the server name, then test the revised function `getNames()` in `fetch-demo.js`. This should succeed and print something like

```
{names:  [ null, 'Eli', 'Eli', 'RAB' ] }
```

on standard output, a single key-value pair in JSON format where the value is an array indicating the names that have been entered so far. The second `.then()` call in `getNames()` assigns this JSON expression (a string) to the javascript variable `data`, and we can extract the array from this JSON expression as the value for the key `names` using the expression `data.names` .

Modify `getNames()` to print this result array in a different format besides a raw JSON string. You can perform a web search to find methods of Javascript arrays.
Some ideas (you could use one or more of these, and/or something else you find):

- The `.forEach()` method for arrays takes a function as its argument and applies that function to each element in that array. You could apply `.forEach()` to the array `data.names` to print each name in that array using `console.out()`, perhaps using a lambda (arrow) to define that function.
- You could print an informational message about how many names were returned using the `.length` property of an array. *Note:* `.length` is not a function – it acts like a state variable.
- The `.slice()` method for an array returns another array that contains a subsequence of the original array's values. You could use `.slice()` and `.forEach()` to print only the first few names, or the last few (with the help of `.length`).

4. Once you have finished and tested your `getNames()` method, create a git `commit` to include the changes in `fetch-demo.js` .
5. Now implement an `addName()` method.
   a) Start with a copy of `addCount()`, and change the copy's function name to `addName()`. This initial copy should behave exactly the same as `addCount()`.
   b) Looking at the code for `addCount()`, we see that there is only one `.then()` call, which prints a success or failure message using `console.out()`. We probably don't need to change that `.then()` call for `addName()`.

```
fetch("http://162.210.90.19:3000", {
      method: "POST"
 })
```

However, the `fetch()` call for `addCount()` has a second argument, which is a Javascript object. The `method` member of that object is used to indicate the type of HTTP request, which is `POST` in this case.
The desired function `addName()` differs from `addCount()` in one crucial respect:

- `addCount()` doesn't need an HTTP parameter (because its server always adds 1), but `addName` needs to provide a name (string) to be added as a parameter. *Our* `/names` *server expects to find that parameter as the* `body` *of the* `POST` *request.*

c) Modify `addName()` as follows:

- Change the URL to `http://162.210.90.19:3000/names`
- Add an argument *str* for the `addName()` function in order to pass the name to be added.
- Add a member

  `body: `name=${str}``

  to the object passed to `fetch()`. This uses the template string `` `name=${str}` `` to produce the string value `'name=`*str*`'` where *str* is the argument for `addName()`. *Notes:*
  - Be sure to use the "backtick" character `` ` `` in that template string expression, not some other kind of quote. Template strings were discussed in class.
  - *Don't* cut and paste the backtick expression above from this PDF document. (We used a special Unicode character to get the correct backtick appearance `` ` ``, and that character might not copy correctly.)
  - **Separate the `body` element from the `method` element with a comma.**
- One more member needs to be added to `fetch()`'s object argument:

  `headers: {"Content-type": "application/x-www-form-urlencoded; charset=UTF-8" }`

  This member informs the `/names` server about the type of the `body`.
  - **Note:** Don't forget to separate elements of that object with commas.

d) Now, test your new version of `addName()`, using a call such as

`fd.addName("`*username*`")`

for your *username*. You can use `fd.getNames()` to test to see that your name was in fact added.

D Delivery

Submit parts A and C by creating a commit and `push`ing to stogit. (Part B submission is part of the exercise.)