CIS*2750
Assignment 3, Module 2

1. Code organization and submission structure

Your Assignment 3 backend will executed as follows:

- Submission is unzipped. If it contains `node_modules/`, then `node_modules/` is removed for you.
- We type "`npm install`" to install all the modules that your assignment needs.
  - `npm` automatically re-downloads and recompiles all the necessary dependencies.
- We run the server using "`npm run dev somePortNum`", where `somePortNum` is one of the port numbers reserved for grading.

Your assignment structure will need to support this. Your assignment must use the A3 stub, which includes both the client and the server stubs. See A3 Stub documentation for details. Since all of your code "lives" on the backend, the entire A3 submission structure is included here.

The submission must have the following directory structure:

- `assign3/` contains `app.js`, `package.json,` and `package-lock.json.` This is also where the `Makefile` must place the shared library file.
  - **NOTE:** remember to delete `node_modules` from this directory. If you don't, CourseLink will most likely prevent you from submitting your assignment due to upload file size limitations.
- `assign3/public/` contains `index.html`, `index.js`, and `style.css`
- `assign3/uploads/` should be empty, but this is where all the .ged files uploaded through the Web client will go.
- `assign3/parser/` contains the `Makefile` that creates your shared library.
- `assign3/parser/src/` contains `GEDCOMparser.c`, `LinkedListAPI.c`, and all other `.c` files
- `assign3/parser/include` contains `GEDCOMparser.c`, `LinkedListAPI.c`, and all other `.h` files

JavaScript

- All of your Module 2 JavaScript functionality must be placed into `app.js`.
- You will be required to delete `node_modules` from the `assign3/` folder before submitting it. In fact, CourseLink will most likely not allow you to submit an assignment containing `node_modules` due to file size limitations.
- You **do not** need any additional JavaScript / Node.js packages to complete the server portion of A3. All the JavaScript / Node.js packages necessary to complete the assignments have been provided for you.
- Please **do not install any additional Node.js packages**. Remember, all your Node modules must be automatically downloaded when we type "`npm install`".
- If you add modules incorrectly, and your A3 backend does not run when we grade it due to missing dependencies, you will lose **all** the marks for Module 2.

C code and shared library

- The source code for your C parser library must be placed into the `parser/` directory of the stub.
- You must include a Makefile that compiles your parser library into a single shared library. Place the Makefile into the `parser/` directory.
- The user must be able to descend into the parser directory and type "`make`" to compile your library.

- Your Makefile must place the shared library directly into `assign3/`, the root directory of A3 - i.e. the directory containing `app.js`.

2. Implementation

JavaScript

Module 2 functionality must support the front-end functionality described in Module 1. As a result, you will need to provide server routes/endpoints - i.e. `app.get()` callbacks and the "paths" that `app.get()` listens for - for the following functionality:

- Getting a list of file stats for the File Log Panel
- Getting a list of stats of a single file for the GEDCOM View Panel
- Creating a GEDCOM file
- Adding an individual to a GEDCOM file
- Getting lists of descendants/ancestors

You will not need to create routes for uploading/downloading of files, since those are already provided for you.

When sending data from the server back to the client, send is as a JSON **object**, not a JSON **string** - i.e. call `JSON.parse()` on the JSON string that you got from a C function, then stick it into the `req` variable of the callback function that you pass to `app.get()`. See the A3 Stub - `app.get('/someendpoint'…)` - for an example of the server responding to a GET request from a client.

The server stub - `app.js` - already accepts a port number as a command line argument. Do not change this, and do not hardcode any port numbers.

C code

The JS code for each route will need to call an appropriate C function for creating/modifying .ged files, or extracting information from them. You will need to write these functions. These functions will have the following general architecture:

- Call `createGEDCOM()` to load data from .ged file - unless this is a function for creating a new .ged file from JSON.
- Extract data from the GEDCOMobject - e.g. get the object summary, a list of summaries about individuals, or run ancestor/descendant searches. Alternatively, you might modify a GEDCOM object by adding an Individual to it.
- If if modifying or creating a .ged file, validate GEDCOM object, then write it to a file. If validate or write fails, return a useful error message or code to JS code.
- Remember to call `deleteGEDCOM()` and free all other memory before returning from the function!
  - While A3 will not be tested for leaks, if your code leaks memory, you might slow down or crash the server, which will just slow you and everyone else down. So be carful with your memory and remember to free your data!
  - If you allocate a string - e.g. a JSON string representing a summary of a GEDCOM object - in the C code, you pass is to JS code and let JavaScript worry about freeing it. However, you must free all dynamically allocated entities that only need to exist while the C function is running.
- return data to JS code, as a JSON string.

All these functions should all be relatively short, because they rely on the functionality that you have already implemented in Assignments 1 and 2.

We will replace this rather inefficient file-based back-end with a database in Assignment 4.

3. Defensive programming

Server-based C code will be particularly difficult to debug. If it crashes - e.g. because you passed the wrong data in a JSON string to a C function - it will do so silently, with little to no output. You will not be able to use gdb or valgrind to debug it. Therefore, be very careful with user inputs and error handling:

- Validate user input in the Module 1 client code
- Figure out what to do with missing data
- Make sure you send correct data from client to server
- Make sure you pass correct JSON strings from JS to C on the server
- Make sure that you correctly parse a JSON string in a C library. You have already implemented most of this in A2. However, the less fault-tolerant your `JSONtoXXX()` functions are, the more careful you have to be when creating JSON strings that you pass to them.