CIS*2750
Assignment 3
Module 2

1. Server functionality and implementation considerations

Server functionality must support the front-end functionality described in Section 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 your following functionality.

The server code will be written in JavaScript and executed using Node.js platform. The Web client will call server routes using HTTP GET requests. The server will interface with the C library. The easiest way to interface will be through strings in JSON format: server will pass JSON-encoded strings to the C functions is calls, and will get JSON-encoded strings as a result.

You will need to write a few extra functions to help C and JavaScript interface better. You can add them to `CalendarParser.h`. Their implementation can be in `CalendarParser.c`. or a separate own file - your call. For example, you can create a C function `createCalFileFromJSON()`, which, given a JSON string encoding a basic Calendar, a string with a file name, and a few additional required bits to make the calendar valid (i.e. an event JSON string and two date JSON strings for the event) will create a Calendar struct, fill out its fields (including a single Event struct), and save it to a file using `writeCalendar()`. You put stitch together the new Calendar from structs returned by `JSONtoCalendar()`, `JSONtoEvent()`, etc..

When a server receives a GET request from client for calendar creation (along with the data), it would call the appropriate C function, pass the data to it, and communicate its return status back to the client.

We already ha a lot of useful functions from A2: creating calendar/event list summaries, adding events to a calendar, etc..

Some general ideas:

1. Think about the information that needs to be exchanged between the UI and the library. You hae almost all the pieces you need from A1 and A2 - you just need to stitch them together.
2. You can finally use the shared library for the calendar parser. JavaScript can dynamically load and call functions from this shared object library using the FFI module.

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

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 your port numbers. Your assignment will be graded with a port number different from yours!

Since we are writing a Web app, the functionality will be stateless. The JS code for each route will need to call an appropriate C function for parsing/modifying/creating Calendar objects and .ics files, or extracting information from them.

You will need to write these functions. These functions will have the following general architecture:

- Call `createCalendar()` to load data from .ics file - unless this is a function for creating a new .ics file from JSON.
- Extract data from the Calendar struct - e.g. get the calendar summary, a list of events, etc.. Alternatively, you might modify a Calendar struct by adding an Event to it.

- If if modifying or creating a .ics file, validate Calendar struct, then write it to a file. If validate or write fails, return a useful error message or code to JS code.
- Remember to call `deleteCalendar()` 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 careful with your memory and remember to free your data!
  - If you allocate a string - e.g. a JSON string representing a summary of a Calendar object - in the C code, you jusr 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 the JS code, as a JSON string.

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

For example, a request from the browser client for a summary of a specific .ics file would have the following flow:

- Client contacts server
- Server calls a C function
- C function opens an .ics file, creates a Calendar object, creates a JSON summary from the calendar object, closes the file/freed the Calendar object, and returns the JSON to the JavaScript caller
- Server passes the JSON to the client, which converts it into an HTML table

We will in part replace this rather inefficient file-based back-end with a database in Assignment 4. However, we still need it, so we can develop a Web-based GUI and populate the database.

2. Code organization and submission structure

Your project backend will executed as follows:

- Submission is unzipped. If it contains `node_modules/`, then `node_modules/`will be removed for you.
- We type "`npm install`" to install all the modules that your assignment needs.
  - `npm` automatically re-downloads and recompiles <u>all</u> the necessary dependencies.
- We run the server using "`npm run dev somePortNum`", where `somePortNum` is one of the port numbers reserved for grading - not the port number assigned to you.

Your project 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:

- `CalendarApp/` 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 before submitting.
- `CalendarApp/public/` contains `index.html`, `index.js`, and `style.css`
- `CalendarApp/uploads/` should be empty, but this is where all the .ics files uploaded through the Web client will go.
- `CalendarApp/parser/` contains the `Makefile` that creates your shared library.
- `CalendarApp/parser/src/` contains `CalendarParser.c` and all other `.c` files
- `CalendarApp/parser/include` contains `CalendarParser.h` and all other `.h` files

JavaScript

- All of your Section 2 JavaScript functionality must be placed into `app.js`.

- You will be required to delete `node_modules` from the `CalendarApp/` folder before submitting it. In fact, Moodle 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**, unless you really know what you are doing. Remember, all your Node modules must be automatically downloaded when we type "`npm install`".
- If you add modules incorrectly, and your JavaScript backend does not run when we grade it due to missing dependencies, you will lose **all** the marks for Part 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 `CalendarApp/` - i.e. the directory containing `app.js`.