CIS*2750
Assignment 4
Deadline: Sunday, November 24, 9am
Weight: 14.5%

1. Introduction

This assignment builds on the GPX parser created in Assignments 1 and 2, and the Web app created in Assignment 3. This component requires you expand your A3 in the following ways:

- Add a UI panel for various database activities (see Section 2)
- Create and maintain tables in an SQL database (see Sections 2 and 3)
- Add connectivity to a MySQL database (see Section 4)

2. Database Design

2.1 Naming conventions

You **must not** change the names of menus, menu commands, buttons, tables, columns, and the like that are specified below. Note that user-specified names in SQL (tables, columns) are case sensitive, but SQL keywords are not. This requirement is intended to simplify and speed up marking and allow, for example, for tables to be prepared with test data in advance. If you change specified names, you **will lose marks**.

2.2 Tables

When your program executes, it must create these tables in your database - but only if they do not already exist. Remember to check for errors when creating tables. If creation fails with the message indicating that the table already exists, you are good to go. However, if creating a table fails for some other reason (e.g. invalid SQL syntax), you must fix the problem!

Another way to check if a table exists is with a select statement:
`select TABLE_NAME from INFORMATION_SCHEMA.TABLES where TABLE_NAME = 'MY_TABLE';`
where `MY_TABLE` is whatever table name you are looking for. If table exists, the query will return one row. Otherwise the result is empty.

Also, make sure that your program **does not drop** any tables. Your code can create, edit, and clear tables, but it cannot drop them. If you need to drop a table, do so manually through the `mysql` command line tool.

We are interested in storing data about files, routes, and points. Thus, the schema for your database consists of three tables named `FILE`, `ROUTE`, and `POINT`. The idea is that every unique file is stored in the `FILE` table. Routes refer to their source files by means of foreign keys, and route points refer to their routes through foreign keys. For the sake of simplicity, we are not worries about track and waypoint data.

Column names, data types, and constraint keywords are listed below. The foreign key constraints ensure that when we delete a GPX file, all of its routes and points are automatically deleted from their respective tables.

Table `FILE`

1. `gpx_id: INT, AUTO_INCREMENT, PRIMARY KEY.` The `AUTO_INCREMENT` keyword gives MySQL the job of handing out a unique number for each organizer so your program doesn't have to do it.

2. `file_name: VARCHAR(60), NOT NULL`. The value of the GPX file.

3. `ver`: `DECIMAL(2,1), NOT NULL`. GPX version for that file. It is almost always 1.1, so the decimal type works.

4. `creator`: `VARCHAR(256), NOT NULL`. The creator for that GPX file (i.e. the tool that created it).

Table `ROUTE`

1. `route_id: INT, AUTO_INCREMENT, PRIMARY KEY.`

2. `route_name: VARCHAR(256)`. Route name. NULL if missing.

3. `route_len: FLOAT(15,7), NOT NULL`. Route length.

4. `gpx_id: INT, NOT NULL.` The file that originally contained this route. FOREIGN KEY REFERENCES establishes a foreign key to the `gpx_id` column in the `FILE` table. Deleting the latter's row will automatically cascade to delete all its referencing routes.

5. Additional constraint: `FOREIGN KEY(gpx_id) REFERENCES FILE(gpx_id) ON DELETE CASCADE`

Table `POINT`

1. `point_id: INT, AUTO_INCREMENT, PRIMARY KEY.`

2. `point_index: INT, NOT NULL`. Index of the point in the original route. The start of the route would have index 0, next point after start 1, etc.. The end point would have index n-1, where n is the number of points in the route.

3. `latitude: DECIMAL(11,7), NOT NULL`. Point latitude.

4. `longitude: DECIMAL(11,7), NOT NULL`. Point longitude.

5. `point_name: VARCHAR(256)`. Point name. NULL if missing.

6. `route_id: INT, NOT NULL`. This is a foreign key referring to the `route_id` column in the `ROUTE` table. Deleting the latter's row will automatically cascade to delete all its referencing points.

7. Additional constraint: `FOREIGN KEY(route_id) REFERENCES ROUTE(route_id) ON DELETE CASCADE`

Note: in a real DB backend, we would never hardcode the route length, since it can change as route points are added/removed from the route. We are including it here for simplicity, so your JavaScript code does not have to re-implement the haversine formula. We will also never modify our tables after the initial insert, so this will not be an issue.

In addition, the points themselves pose a bit of a problem. On the one hand, the SQL decimal type is perfect for storing latitude / longitude of a point, since this type is capable of storing it precisely, and does not suffer from the float/double precision errors. On the other hand, table rows are unordered, and point lists must be ordered to form a route. Therefore, we have added the index of a point in the original GPX file to the `POINT` table. This is a hack, but it is easy to implement and will give us what we need.

3. Database functionality and UI additions

To interact with the GPX database, add a **Database** UI section with the items described below. This section can be placed on the Web page below the A3 functionality. You can provide a separate sub-header for A4 section of the Web interface, to make it more visible. A proper Web app would use a less clunky solution, but we are trying to keep things simple.

Each UI item is only active when it is logically meaningful. For example, we cannot run queries if the tables are not created. We also cannot run queries if there are no GPX files on the server. After every command, except **Execute Query**, display up an alert with the status line based on a count of each table's rows, e.g.: "Database has N1 files, N2 routes, and N3 points" (how you get this information is up to you).

- **Login**: Your UI must ask the user to enter the username, password, and database name, and will attempt to create a connection. If the connection fails, your program must display an error (as an alert) and prompt the user to re-enter the username, DB name, and password.
- **Store All Files**: This command is used to insert all the data from the files displayed in the File Log Panel into your database. This is active / visible only if the File Log Panel contains at least one file - i.e. there is at least one valid GPX file on the server in the `uploads/` directory. For every file, go through the following steps:
  - Obtain all necessary data that should be stored. By now, you shouldn't have to write any new C code, and can just use the functions from Assignments 2 and 3.
  - Check if the file name is already in the `FILE` table. If not, insert a new record for this file into the `FILE` table, and add the appropriate records for its routes and points into the `ROUTE` and `POINT` tables. Because of the foreign key constraints, you have to do this in the appropriate order. Use `NULL` for any missing fields.
  - Keep in mind that you cannot have routes with no files, or points with no routes, so if you ever end up with an empty `FILE` table and non-empty `ROUTE` - or empty `ROUTE` and a non-empty `POINT` - something definitely went wrong!
- **Clear All Data**: Delete all rows from all the tables. This may have to be done in a specific order due to the foreign keys. This is only active if tables have data in them. Do not drop the tables themselves.

  You do not need to update the File View Panel, or GPX View panel. That A3 functionality does not rely on the database, and does not need to be modified.
- **Display DB Status**: This displays an alert with the status line described above: "Database has N1 files, N2 routes, and N3 points". Again, how you get this information is up to you - there are multiple SQL queries that you can use here.
- **Execute Query**: this UI item is used to query the database. It should contain a way to submit one of the five standard queries discussed below. The results from the submitted query are displayed here in a scrollable table.

Queries

The **Execute Query** menu item displays four standard queries, with some parameters that the user may fill in. There should be some way to select which query is submitted. The queries must be displayed in simple English, but your program will generate the underlying SQL statements incorporating any filled-in variables.

The intended user of this functionality is someone who wants to access a GPX file database (or just a database of routes), so think about what they might want to know, and how to make it easier for them to provide the information necessary to execute the query.

The result of the each query must be displayed as a table. The required queries are given below.

Queries:

1. Display all routes. Your UI must allow the routes to be sorted either by name or by length.

2. Display the routes from a specific file. This query displays the names and lengths of all routes, as well as the file names. Allow the user to sort the results by file name, route name, and route length.

3. Display all points of a specific route, ordered by point index.

4. Display all points from a specific file. Order them by route name. If routes have no names, you can order them in as you see fit, and give routes unique labels, e.g. "Unnamed route 1", "unnamed route 2", etc.. For every route, individual points need to be ordered by point index.

Helpful hints:
- You can use SQL to sort query results in the SQL query itself (lecture 15a). You can also do it on the JavaScript side - remember, a call to `connection.execute` returns an array of row data, which can be sorted using a predicate function. Pick whatever you think is easier for you.
- You might find that some queries can be done using either a single SQL complex statement, or multiple simple statements - e.g. run SQL statement 1, and based in its results run SQL statement 2. Again, pick whatever you think is easier for you to do.
- Make sure you carefully review Lecture 15b as you implement your new `app.js` routes, to make sure you use the Node.js `mysql2` package correctly. Pay close attention to the use of `await` keyword, especially you need to sequence multiple queries in the same function.

4. Connecting to the SOCS database

4.1 Connection details

Our official MySQL server has the hostname of `dursley.socs.uoguelph.ca`. Your username is the same as your usual SoCS login ID, and your password is your 7- digit student number. A database has been created for you with the same name as your username. You have permission to create and drop tables within your own named database.

To access the database from home, connect to the school network using a VPN (Cisco AnyConnect), and

- Login from home using the `mysql` command line tool, if assuming you have it installed in your machine or VM. See Lecture 15b for details.

- Execute JavaScript code with SQL queries from your machine and look at the output.

You can also login to linux.socs.uoguelph.ca, and use the `mysql` command line tool from there (it is installed)

Week 9 and 10 notes and examples have details for how to connect to a MySQL server, create/drop tables, insert/delete data, and run queries. This includes simple JavaScript programs that connect to the SoCS MySQL server and run some queries .

4.2 Implementation details

The A4 solution must be added to your `GPXApp/` directory.

All the new UI functionality goes into `index.html` and `index.js`. All code for connecting to the DBMS server and interacting with the database must be placed into `app.js`. Your GUI client will interact with the server to load the data and run the queries. You will need to create additional endpoints on the server for this.

You will need to install the `mysql2` for Node.js: `npm install mysql2 --save` . Make sure you include the `--save` flag - it will automatically update `packages.json` to include a reference to the `mysql2` package. Again, remember that you must submit A4 without the `node_modules` directory, and we will install the Node modules by typing `npm install`.

You will develop your assignment using your own credentials and database, but part of the grading process will involve connecting your code to our database. This means that you cannot simply hardcode your credentials into

`app.js`. As mentioned in Section 2, your UI **must** ask the user to enter the username, password, and database name, before it attempts to create a connection.

5. Grade breakdown

- Correct database tables and database connection, obtaining the user's credentials: 15 marks
- Usability of UI additions, including error handling: 27 marks
- **Store All Files**: 20 marks
- **Clear All Data**: 5 marks
- **Display DB Status**: 5 marks
- **Execute Query**, fully functional: 28 marks
  - Required queries: 4 @ 7 marks each

The UI will be expected to connect to a functional database backend.

Deductions

Marks will be deducted for buggy or broken functionality. If the UI is not connected to the database backend, only minimal marks will be given for the UI itself. The usability marks will apply only if the UI is connected to the database.

Usability

Your assignment is expected to comply with the usability principles that are covered in lectures 18-19. Examples include, but are not limited to:

- Making sure the UI is consistent, and does not break the user's expectations
- Handling user errors in the UI
- Making UI output readable and formatted for a general user - error messages are not just raw SQL/ JavaScript dumps, etc..
- Making sure that the UI, where possible, prevents the user from performing invalid actions, or entering invalid data. For example, if a set of valid values is known in advance (e.g. file names), present them as a drop-down list, instead of forcing the user to type them in - and potentially allowing them to make mistakes.
- Using colours and fonts that are legible - avoid small fonts, make sure there's enough contracts between background and text, etc..
- Making sure the user always knows what happened, for any success or failure of a query - i.e. don't just display nothing if, for example, the query returns no results.

When in doubt, use the UI principles from lecture 19.

As always, test your code on the SoCS servers. As with A3, the server for A4 must be developed and tested on is `cis2750.socs.uoguelph.ca`. This is where we will run it. The Web app will be accessed from Firefox on `linux.socs.uoguelph.ca`.

Any compiler errors will result in an automatic grade of **zero** (**0**) for the assignment.

Additional deductions include:

- Any compiler warnings: **-15 marks**
- Any additional failures to follow assignment requirements: **up to -25 marks**
  - This includes creating an archive in an incorrect format, having your Makefile place the `.so` file in a directory other than `GPXApp/`, modifying the assignment directory structure, creating additional `.js` and `.html` files, etc.

6. Submission

Submit all your C and JavaScript code, along with the necessary Makefile. File name must be `A4FirstnameLastname.zip`.

Late submissions: see course outline for late submission policies.

<u>This assignment is individual work and is subject to the University Academic Misconduct Policy.</u> See course outline for details)