COMP 10222 - Emerging Web Technologies

Lab #1 - REST API with NodeJS

Due date: Friday September 25th at 11:59pm

Worth: 14% of total grade

Marks: 100 marks total

Learning objectives

- Create a REST API with NodeJS
- Use an SQLite database to read, create, modify and delete entries
- Create HTTP requests in NodeJS to test a REST API

Requirements

We will create a basic REST API with NodeJS and write an application to test the API. The REST API will allow for the management of a collection of users, with the functionality of the API defined below.

Server

Create a NodeJS application file named **server.js**. You will define your REST API in this file. Use Express to create a server and define routes as necessary to implement the API. Use an SQLite database table to store and manage the collection of users. The server should run on the localhost, and be accessible at port **3000**.

Database and Users table

The users table should contain the following fields:

- msgid - auto-incrementing integer **(or you can just use the standard 'rowid' that SQLite automatically creates...)**
- status - text
- message - text
- timestamp - date and time (up to you how you implement it, there are three ways, see documentation)

The **server.js app** should itself create the SQLite database and the table. The SQLite database should be stored in a file named **api.db**. The table should initially be empty (i.e. do not populate it with anything when the server initially begins running). If the database file and/or table already exists, they should be wiped out (e.g. delete all previous entries in the table) at the start of the application's execution to ensure that the table is initially empty.

REST API

The REST API should be accessible at **http://localhost:3000/api/**. Implement the following http methods for the collection (i.e. the table) and individual items (i.e. the users). This pattern of methods and functionality is the standard for REST APIs.

Collection

- **GET request** to **/api/** should return a JSON array of the entire collection (which could be empty).
- **PUT request** to **/api/** should contain a JSON array in the request body representing a new collection of items. The current collection should be ***replaced*** with the new collection. The PUT request should return "REPLACE COLLECTION SUCCESSFUL" in the body.
- **POST request** to **/api/** should contain a JSON object in the request body representing a new item. The new item should be added to the current collection. The POST request should return "CREATE ENTRY SUCCESSFUL" in the body.
- **DELETE request** to **/api/** should delete the entire collection. The DELETE request should should return "DELETE COLLECTION SUCCESSFUL"

Item

- **GET request** to **/api/id** should return a JSON object of the entry in the collection with the supplied id.
- **PUT request** to **/api/id** should contain a JSON object in the request body representing updated values for the item with the supplied id. The item should be updated in the current collection to reflect these new values. The PUT request should return "UPDATE ITEM SUCCESSFUL".
- **DELETE request** to **/api/id** should delete the item from the collection. The DELETE request should should return "DELETE ITEM SUCCESSFUL".

Automated Test Application

Write a NodeJS application called **client.js** that will test the REST API using a series of http requests. Implement the application using

the NodeJS http package (i.e. **var http = require('http');**), or with another package of your choosing, such as axios.

The test application should run through a series of tests in sequence and check that all tests have passed. If all tests have have passed it should print to the console "ALL TESTS SUCCESSFUL". Assume that the test application will only be run once before the server is reset (i.e. it will not be run twice after the server has been started, and it doesn't need to function correctly if it were to be).

The test application will need to setup and make HTTP requests to the server / API. The test application will need to check the response body of requests for the expected information (either a successful response, or the correct item(s) from the collection).

The test application should run through the following tests:

- **Test #1**
  1. Execute three POST requests to insert three items into the collection.
  2. Execute a single item PUT request to modify a single item in the collection.
  3. Execute three separate item GET requests to check if each item is correct.
- **Test #2**
  1. Execute a single collection PUT request that replaces the collection with 4 new items.
  2. Execute a single collection GET request to check if all the items are correct.
  3. Execute a single item DELETE request to delete a single item from the collection.
  4. Execute a single collection GET request to check if all the items are correct.

- **Test #3**
  1. Execute a single collection DELETE request to delete the entire collection.
  2. Execute a single collection GET request to check if the collection is empty.
  3. Execute a single collection PUT request to replace the collection with 3 new items.
  4. Execute two POST requests to insert two items into the collection.
  5. Execute a single item PUT request to modify a single item in the collection.
  6. Execute a single item DELETE request to delete a single item from the collection.
  7. Execute a single collection GET request to check if all the items are correct.

Testing

It may be helpful to view the contents of your SQLite database file (i.e. the equivalent of phpMyAdmin for MySQL). There are many freely available browsers, some you can download and install, but given that we're not doing anything too complex you may find it easiest to just use a cheap and free web viewer.

There are also tools that allow you to send diffent kinds of HTTP requests, for example: Advanced REST Client. These may also help you to test your server code!

Example

A PUT request to /api may include a JSON object such as this in the request body:

[{"status":"Warning", "message":"Overheating machine","timestamp":"2019-09-03T15:36:56.200"}, {"status":"Error", "message":"Coolant leak","timestamp":"2019-09-04T03:45:23.600"},{"status":"Critical", "message":"Explosion imminent","timestamp":"2019-09-05T06:25:23.600"}]

Then we would expect the API to send back in its response body: "REPLACE COLLECTION SUCCESSFUL".

Hints

To create the SQLite database from inside the **server.js** file, you can check if a file named **api.db** exists (try **fs.existsSync**). If it does not exist, the server can create it (try **fs.openSync**).

You may find these sources and examples helpful:

- SQLite NodeJS
- NodeJS - Express Framework
- How do I make an http request?
- Anatomy of an HTTP Transaction

And of course there's the official documentation for SQLite, Express, and NodeJS.

Submission instructions

When you have completed the lab, put your source files in a folder called **lab1.zip** and submit it on the Dropbox.

Failure to follow submission instructions may result in a mark deduction up to receiving a mark of zero on the lab.

Statement of Authorship

Copying others people work and claiming it as your own is a serious breach of ethics. If copied work is found, all parties involved will receive a failing grade as per departmental policy. Group work is encouraged but it is expected that you do your own work. If you do, you'll learn the material and feel better for it. Since all work submitted is assumed to be your own original work, you must include the following "Statement of Authorship" in every program file you submit for grading:

"StAuth10065: I John Doe, 123456 certify that this material is my original work. No other person's work has been used without due acknowledgement. I have not made my work available to anyone else."

- Use the exact text above. No substitutions.
- Replace John Doe with your name and the number 123456 with your student ID.
- Place this text as one line as close to the top of each document as possible.
- Include StAuth10065: at the beginning of each statement.
- Failure to include this statement will cause your work to be ineligible for grading.

Marking scheme

Server	5
SQLite	10
Routing	15
REST API	20
Test application	50
Total	/100

Labs submitted late will receive a 15% penalty per day. Labs more than 2 days late will receive a grade of zero. For example, a lab submitted at 11:01pm on the due date that was marked 80/100 would receive a mark of 80 - 15 = 65/100. A lab submitted at 11:01pm the day after the due date that was marked 65/100 would receive a mark of 65 - 30 = 35/100. A lab submitted at 11:01pm two days after the due date would receive a mark of zero.