Middlesex University London

Coursework 2 (CW2): Node.js Server and Express API (30%)

Deadline: Week 15.

Type of CW: Individual Work.

IMPORTANT. **A submission could receive zero marks if it fails any of the following requirements:**

- The student must **book an available slot for the demo** of a CW**, by the CW submission deadline,** via the related "Individual Demonstration Booking System" on the Module Page, **AND** the work must be **demonstrated within the demonstration weeks scheduled,** during the **demo slot booked** (**IMPORTANT**: see more details and guidance on demo booking at **section 7.4.4 in the handbook**, and for guidance on other important related aspects at **section 7.4 in the handbook**).
- The work must be **demonstrated** and **explained satisfactorily during demonstration**, i.e., student can explain what the code does.

Task. The goal of this coursework is to build the Back-End for the app created in the first coursework. MongoDB will be used for storing the data, and data exchange between the app and the database will be done through REST API implemented using Express.js. The Back-End will run on a Node.js server.

Submission

Instructions for the text area to compile when uploading your submission:

- please, indicate the same information requested below for the README file

Instructions for the zip file, it must contain the following elements, and must be no more than 10MB:

- your code in **2 folders** respectively related to:
  - your **Vue.js App**.
  - your **Express.js App** (**do not include the 'node_modules' folder**, otherwise, the zip will be too big to submit).
- a README file with the following links:
  - [**Vue.js App**] the link to your GitHub Repository. ○ [**Vue.js App**] the link to your GitHub Pages from where the app can directly run.
  - [**Express.js App**] the link to your GitHub Repository. ○ [**AWS Express.js App**] the link to the AWS route that returns all the lessons.

- the **'lesson'** and the **'order' collections** exported from your MongoDB Atlas. See here for how to export collection in MongoDB Compass (https://docs.mongodb.com/compass/current/import-export#export-data-froma-collection).
- the requests you created in Postman. See here for how to export requests in Postman (https://learning.postman.com/docs/getting-started/importing-andexporting-data/#exporting-collections).

Overall Requirements

A submission could receive zero marks if it fails any of the following requirements:

- the Back-End server must use "Node.js"; others such as Apache or Xampp are not allowed.
- in relation to hosting the Back-End server, it not allowed to use AWS S3, nor AWS EC2, nor other cloud-based hosting solutions (e.g., Heroku). The only allowed one is AWS with the technologies covered in the related Lecture.
- the REST API must be developed with "Express.js".
- the Front-End data access must be achieved with "promise" using "fetch" function; "XMLHttpRequest" or library such as axios:js are not allowed.
- the data must be stored in "MongoDB Atlas" and retrieved via your Express.js App; local MongoDB or any other databases are not allowed.
- connection to MongoDB (in your Express.js App) must use the native Node.js driver only; libraries like Mongoose are not allowed.

Marking criteria

- **General Requirements (8%):**

  A. [GitHub Repositories] the code of the Vue.js App must be hosted in a GitHub repository (if you want, you can continue using the one of CW1), and the code of the Express.js App must be hosted in another GitHub repository (2% if 2 separated repositories are used, 1% if only 1 repository is used).
  B. [GitHub Pages] the Vue.js App must be hosted and demonstrated on/via GitHub Pages and connected (via Fetch) to your AWS Express.js App (3% if requirement is fully covered, 1% if app is running locally).
  C. [AWS] the Node/Express server must be hosted on Amazon AWS (https://aws.amazon.com/) (3% if requirement is fully covered, 1% if the server is run locally, 0% if the server is hosted in another cloud-based solution).

- **MongoDB should have (4%):**

  A. a collection for lesson information (minimal fields: topic, price, location, and space) (2%).
  B. a collection for order information (minimal fields: name, phone number, lesson IDs, and number of space) (2%). Suggestion: the element lessonIDs can contain 1 or more lesson IDs, depending on how many different kinds of lessons you have in your order. Other solutions could be accepted: in fact, how you design this is not the primary aspect of this module, therefore it is up to you to find a reasonable solution that works satisfactorily.

- **Middleware Functions implemented in the Express.js Server should include (4%):**

  A. a "logger" middleware that outputs all requests to the server console (2%).
  B. a static file middleware that returns lesson images, or an error message if the image file does not exist (2%).

- **REST API implemented in the Express.js Server should include (6%):**

  A. one GET route /lessons that returns all the lessons as a json (1%). Example:
  [
  {' topic': 'math', 'location': 'Hendon', 'price': 100, 'space': 5},
  {' topic': 'math', 'location': 'Colindale', 'price': 80, 'space': 2},
  {' topic': 'math', 'location': 'Brent Cross', 'price': 90, 'space': 6},
  {' topic': 'math', 'location': 'Golders Green', 'price': 95, 'space': 7},
  ]
  B. one POST route that saves a new order to the "order" collection (2%).
  C. one PUT route that updates the number of available spaces in the "lesson" collection after an order is submitted (1%).
  D. at least one Postman request is created for each route, and the student is able to test all of them properly and explain them (2%).

- **Fetch Functions implemented in the Front-End should include (3%):**

  A. one fetch that retrieves all the lessons with GET (1%).
  B. one fetch that saves a new order with POST after it is submitted (1%).
  C. one fetch that updates the available lesson space with PUT after an order is submitted (1%).

- **Search Functionality (5%):**

- [Intro] this is the challenge component of this coursework, and it is not expected that everyone can complete it. The solution is not covered in the lecture or lab, so you need to research it.
- [Feature Description] The goal is to add a full-text search feature, similarly to the challenge component of the Coursework 1.
- The user can search for a lesson without specifying which attribute to search on.
- The search should send back results that include both "title" or "location", while it is not required that it works also for “price” and “availability”.
- [Difference with CW1 Search Functionality and Development Strategy/Constraints] The difference with “CW1 Search” is that the search here needs to be performed in the Back-End (Express + MongoDB), not in the Front-End as in CW1 (where it was implemented via Vue + JavaScript). You will not receive any mark, for this part, if the search is performed in the Front-End.
- You cannot use any existing library to implement this function. Otherwise, you will not receive any mark for this part.
- Solutions provided are marked as follows.

A. “Fetch” (2%), in the front end, a “fetch” request should be created to send the search information to the Back-End.
B. “Express API” (2%), an Express.js route should be created to handle the search request, and to return the search results from the MongoDB. The student should implement this as a GET route (“/search”), and should be able to test it also without using the Front-End.
C. [Further Point] “search as you type” (1%), similarly to Coursework 1, there is also this mark if the search supports “search as you type”, i.e. the search starts when user types the first letter (displaying all the lessons containing that letter), and the result list is filtered as more search letters are entered (similar to Google Search).