Assignment 9

CS 290 (some sections. Winter 2023. This file was created on Feb 26. See Canvas for clarifications.

Introduction

In this assignment, you will use the MERN stack to add a Single Page Application (SPA) to your existing website/app. The topic of this new page is up to you, but it must interact with a MongoDB cluster based on a Mongoose Schema.

This is the "Portfolio Assignment" for the course. This means that you are allowed to post the entirety of the assignment publicly (e.g., Github, personal website, etc.) *after the term ends*. You can use this as an opportunity to showcase your work to potential employers. Be sure to style your work uniquely...not like the demos.

Note: The material you need to know to complete this assignment is covered in these Explorations:

- Module 1, 2, 3 (HTML)
- Module 4 (CSS and global design features)
- Module 7, 9 (React)
- Module 8, 9 (MongoDB, Mongoose, REST)
- Use the Movies app starter code from Module 9; ***not 8***.

Be sure to periodically review Assignment 9 Tips thread in the Ed discussion board (pinned at the top)

Learning Outcomes

- Choose a topic that is suitable for "logging" data in a table.
- Add an SPA page to the existing website to interact with MongoDB.
- Define a schema that includes string, number, and date types.
- Perform Create, Retrieve, Update, and Delete on the database using the existing React frontend and new REST backend.
- Use a component to render one row of data from the log.
- Use map() to render rows of data from the log.
- Create a new home page that describes the technologies used in the website.
- Move the existing Web Dev topics from the home page to a new page of the site.
- Update the global navigation.
- Improve the frontend design with changes in spacing, color contrast, and the addition of helpful icons.

Tools

Existing tools, such as the React framework v18, React icons, Node Fetch, React Image Gallery, and React Router DOM. Use either a MongoDB Atlas Cluster account, or a local installation of MongoDB. Incorporate Mongoose and REST API. Continue to use Node.js v18. Continue to use LanguageTool to check grammar, spelling, and punctuation.

Instructions

This Assignment will require use of your existing React frontend website that has pages, components, styles, icons, and images for Home, Gallery, Staff, and Order. Choose a topic for data to log using that frontend. Connect the frontend to the backend, which uses MongoDB, Mongoose, and REST.

Topic

In this course, you have experimented with object data, such as products, people, books, and movies. The demonstration for this Assignment experiments with exercise data. Now is your chance to develop a set of data that has meaning to you. Choose a new topic that logs data which incorporates string, number, and date types. Here are some ideas:

- Tasks you completed (timesheet).
- Jobs you've held.
- Applications for internships, jobs, scholarships, grants, etc.
- Places you have travelled to.
- Inventory of collectables.
- Inventory of your belongings (for insurance purposes).
- Medical procedures.
- Car mileage.
- Gaming milestones.
- Stock Trades.
- Gardening tasks.
- Home Repairs.
- What else can you think of that uses strings, numbers, and dates?

Keep it simple! You can always embellish the work after the term ends.

Structure of folders and files

Set up the new Assignment for success:

1. In your existing Module 9 folder, add a new folder called ***a8-username-portfolio***.

2. Place a copy of your existing React website from the previous Assignment 7 into the new portfolio folder, and rename it "frontend". It must continue to have its own package.json, node_modules, and .env file.

1. Open the .env file and change the port to 8000. If, for some reason you did not have this file in the previous assignment, add it now.
2. In VS Code, right-click on the new frontend folder and choose **Open in Integrated Terminal**. This will allow you to operate the frontend separately from the backend.
3. The frontend of your site will now display in the browser at this address: http://localhost:8000/.

3. In the new portfolio folder, add a folder called "backend". This folder will hold the model.mjs and controller.mjs files needed to interact with MongoDB. The backend folder will have its own package.json, node_modules, and .env files.
    1. In VS Code, right-click on the new backend folder and choose **Open in Integrated Terminal**. This will allow you to operate the backend separately from the frontend.
    2. In the VS Code terminal, initialize Node with npm init and provide your topic name, description, and your name as the author.
    3. Install these dependencies using npm: dotenv, express, mongoose, nodemon, and rest
    4. In the new package.json file, add the start script in the "scripts" section: "start": "nodemon controller.mjs". To the root of the backend folder, add a .env file. In this file, add your MongoDB connection string and port number:

        MONGODB_CONNECT_STRING='mongodb+srv://YourAccountName:YourAccountPassword@cluster0.ybrfb.mongodb.net/test'
        PORT=3000
        Replace the account and username in the string to match your Atlas Cluster. Use the connection string provided by your account; it may not match this string exactly.

        OR, if you have MongoDB working in your local hard drive, you can use this string: mongodb://localhost:27017/

    5. To the root of the backend folder, add a file called model.mjs.
    6. And, add a file called controller.mjs.

Backend Model and Schema

Following the format of the movies-api-backend.zip and videos, create a similar set of scripts for your new topic.

1. Import mongoose and dotenv/config

2. Connect to mongoose using the .connect() function.

3. Define a database variable, so it uses that .connect() function.

4. Use that variable to connect to the database using .once().
    - Use if to provide a res.status(500).json message if the server isn't connecting.

    - Else, provide a `console.log()` message that states the server is connected. Be specific about which collection you are connected to.
5. Define a variable for the schema object using `mongoose.Schema()` The schema must include at least one `string` type, one `number` type, and one `date` type.
    - Each property must be `required`.
    - You may also add `default` values and `min` or `max` values.
    - For the `date` property, incorporate: `default: Date.now`. More about dates later.
6. Below the schema, define a variable for a *document*. It must pass in a string for the `"name"` and the schema name you defined above.

7. Define a model to **create** a document in that new collection, using `async`, `new` with {*params*}, and returns using `save`.

8. Define a model to **retrieve** documents from the collection using `async`, `.find()`, and `return .exec()`.

9. Define a model to **retrieve** a document by `id` using `async`, `.findById(id)`, and `return .exec()`

10. Define a model to **update** a document using `async`, `await`, `.replaceOne({params})`, and `return {params}`.

11. Define a model to **delete** a document using `async`, `await`, `.deleteOne({id:id})`. The `return` will result in `deletedCount`.

12. Export all of those model names in an array. The model function names you export will be used in the controller.mjs file.

Backend Controller

Continue following the format of the movies-api-backend.zip and videos, create a similar set of scripts to work with your models.

1. Import `dotenv/config` and `express`.

2. Define a collection name and import everything as that collection name from the model file.

3. Define the PORT using .env.

4. Define the app to call `express()`.

5. Define a **Create** route using `app.post()` with a path that matches the path you'll use for your frontend's new page. For now, use `'/log'` for the path (which is a generic-enough location for any topic). In the route, use the collection name to call your model function for create. Pass in

all the schema's properties. Use `.then` to send status `201` if the creation was successful and `.catch` to send status `400` if there was a user-input error. Write the error message in complete sentence to make it clear what occurred.

6. Define a **Retrieve** route using `app.get()` with the same path as for create. In the route, use the collection name to call your model function for retrieve. Use `.then` and an `if` statement to determine if the document exists, and if so, respond with JSON and that document name. Else, respond with status `404`. `.catch` to send status `400` if there was a user-input error. Write both error messages in complete sentences to make it clear what occurred.

7. Define an **Update** route using `app.put()` with the same path. In the route, use the collection name to call your model function for update. Pass in all the schema's properties. Use `.then` to respond with JSON and that document name and `.catch` to send status `400` if there was a user-input error. Write the error message in complete sentence to make it clear what occurred.

8. Define a **Delete** route using `app.delete()` with the same path but with :id appended: `/log/:id`. In the route, use the collection name to call your model function for delete. Pass in the schema's `id` property. Use `.then` and an `if` statement to determine if the document was deleted, and if so, respond with status `204` and `.send()`. Else, respond with status `404`. `.catch` to send status `400` if there was a user-input error. Write both error messages in complete sentences to make it clear what occurred.

9. Define an `app.listen()` route to provide a `console.log` message telling you which port the backend is listening to.

Testing

If your connection to the backend is not active, right-click on the backend folder and choose **Open in Integrated Terminal**.

Test the use of your new schema and CRUD operations by creating requests in the browser address bar, like you did for Module 8's examples.

Retrieve User Interface

Now we bring our attention back to the frontend React framework. Following the format of the movies-api-frontend.zip and videos, create a similar set of functions for your new topic.

The functionality of this page is to retrieve all the documents from your collection and display them in a table. The table will include icons to Create, Edit, and Delete documents from the collection. In your frontend folder, create a page for retrieval of documents:

1. In the src < pages folder, make a new page and name it something related to your new topic; something like **LogPage.js**.

2. In the LogPage.js file, add imports for `React`, `useState`, `useEffect`, and `useNavigate`.

3. Define a function with the same name as your page, for example: `function LogPage({}) {...`
    - Define a variable to take advantage of `useNavigate()`.
    - Define a variable to incorporate the document name and set that document equal to `useState([])`.
    - Define a variable to **retrieve** the log of documents using `async`, `await`, and `fetch` methods. Retrieve from the path you defined in the controller.mjs file: `/log`. Define the response variable. Use `setDocument(collectionName)` to hold the data.
    - Define a variable to **update** one document using `async` and `set` methods. Use `navigate` to define the endpoint path where the Edits will take place. This path will also get used with an icon to edit a single document. More about that later.
    - Define a variable to **delete** one document by `id` using `async`, `await`, and `fetch(`/log/${id}`, { method: 'DELETE' })` methods. Use an `if` statement with status `204` that fetches the `/log` and responds with JSON and `set`. Else, respond with the `id` and status code. Incorporate `useEffect()` to load all the documents.
    - Return the HTML needed to create the user interface, such as `<h2>Page Name</h2>`, `<article>`, descriptive paragraph, and a component* that calls in the collection name, delete function, and edit function.
    - Close the return and export the function.
4. * In the existing components folder, create a file with a function that creates a **table**. The table must ...
    - Import an icon for the Create function.
5. Call in the collection name, the delete function, and the edit function.

6. It must return HTML for a table, caption, thead, and tbody.

7. The `<caption>` must provide the icon that allows users to Create a document.

8. The `<tbody>` must use the collection name to `map()` each document by index.

9. The `map()` function must reference a component** for one row/document of the collection.

10. ** In the existing components folder, create a file with a function that creates a **row** for the previous table. The row must ...
    - Import icons for Update and Delete functions.
    - Return a `<tr>` the number of columns to match your schema, in addition to holding Delete and Edit icons with functions, as specified in the `LogPage()`.
    - `<td>`s will call in the collection name with parameters appended, like this: {log.name}

    - When calling the date, `.slice()` the default date format, so it displays just the first 10 characters (this excludes the timestamp), like this: `{log.date.slice(0,10)}`
    - Close the return and export the function.
11. Revise the imports on these files to ensure that:
    - The *table* component imports the *row* component.
    - The LogPage imports the *table* component.
    - Call its function in the HTML using its component tag/function name. It must call in the variable name for the response (noted above).

Testing

If your connection to the localhost is not active, right-click on the new frontend folder and choose **Open in Integrated Terminal**.

Test the page in the browser at `http://localhost:8000/`. At the very least, the page heading and table should render. Once all the components and pages are working smoothly with the backend, the data should render on this Log page.

Create User Interface

In order to immediately display new and existing documents in our React log page, we need to input the data, then upon clicking send, the data is added to our collection based on the schema. It is then retrieved while we navigate back to the log page.

The create page should use a table that visually matches the retrieval table (to avoid user confusion).

1. In the src > pages folder, make a new page to create new documents for the log, something like: **CreatePage.js**.

2. In the CreatePage.js file, add imports for `React`, `useState`, and `useNavigate`.

3. Export and define a function with the same name as your page, for example: `function CreatePage({})`
    - Define a set of variables to `set` and `useState()`. Feel free to add default values in a string to one or more of these variables. For example, if the first param is the name of an exercise, then the default can be `useState('Walking');`
    - Define a variable to take advantage of `useNavigate()`.
    - Define a variable that adds to the log using `async()`, and:
        - Defines a variable to create a single document, specifying each param.
        - Defines a variable to `await` and `fetch('/log')` with the `method:`, `body:`, and `headers:`.
        - Uses `if` to check for status `201` and provides an `alert` with a success message in a complete sentence. Else, it sends an `alert()` with failed message that makes it clear what occurred.
        - Calls the `navigate` variable defined above and reference the path `/log`.

        - Returns the HTML needed to create the user interface, such as the page name, article, descriptive paragraph, and table.
        - The table should use `<label>`s and `<inputs>`s and/or `<select>`s to take user input for the document parameters.
            - If a schema param is of type `Number`, then the form control is `<input type="number" />`.
            - If a schema param is of type `Date`, then the form control is `<input type="date" />`.
        - Form controls must specify the `value={param}` and `onChange={e => setParam(e.target.value)}`
        - The `<button>` to save the entry will use `onClick` for the function name.
    - Close the return and export the function.
4. Revise the LogPage to import the new Create page. The component tag/function does *not* call in the Create parameters.

Testing

Test the page in the browser at `http://localhost:8000/`. With the frontend and backend terminals active, use the new form controls to add a document to the collection. Does the data get added and does it show up on the LogPage?

Edit User Interface

In order to update a document and immediately display it in our React log page, we need to populate the form controls with the existing data. Any or all of the data for a single document can be updated in the Edit page. Upon clicking send, the data is revised in the collection of documents, then retrieved while we navigate back to the log page.

The edit page should use a table that visually matches the retrieval and create tables (to avoid user confusion).

1. Copy the CreatePage.js and rename it EditPage.js (or something similar).

2. Imports are the same for Edit as they are for Create.

3. Change the function name to match the file name.

4. The function should pass in a name for the parameters (as defined in the LogPage.js function).

5. Use that name to append the .params to in the `useState`, like this:
   `const [name, setName] = useState(exercise.name);`

6. Update the variable names to refer to *editing*.

7. Update the `fetch()` value to include the `.id` param. For the body of the fetch, `stringify()` each of the params. For example: `name:name,`.
    - Uses `if` to check for status `201` and provides an `alert` with a success message in a complete sentence.
    - Else, it sends an `alert()` with failed message that makes it clear what occurred.
8. In the return, update the HTML heading and paragraph to reflect the purpose of the page (editing).

9. For the save button, refer to the function name for edit (rather than create).

10. Close the return and export the function.

11. Revise the LogPage to import the new Edit page and
    - Call in the Edit parameters in the component tag/function.

    - In addition, call in the Delete parameters in the component tag/function. Its function exists in the LogPage.js file already.

Testing

Test the page in the browser at `http://localhost:8000/`. With the frontend and backend terminals active, use the new edit controls to update an existing document in the collection. Does the data get updated and does it show up on the LogPage?

Also delete a row to see how that behaves. The action should be instantaneous.

Global Updates

The last few changes to your portfolio site include improving the home page, moving the existing writing to a new page, updating the global navigation, and improving spacing and readability/legibility of the text.

You may also embellish the site with icons (or other imagery) to improve user experience.

1. In the src > pages folder, add a new file called TopicsPage.js (or something similar). Cut and paste your existing writing about Web Dev topics into the `return` area of this new function/page. Be sure to also include your local navigation, if any. If you did not style the local navigation using `display:flex`, now is a good time to do so. If the menu is too long to fit nicely across the article/section/main space, then `flex` will solve the problem.

2. Now that the HomePage is empty of content, give it a new, welcoming heading and a paragraph or two that explains all the technologies and methods you incorporated into the

website. Feel free to use lists and anchors to link to other supporting websites and your social media accounts.

3. Update the Nav.js component to include the LogPage and TopicsPage.

4. Use LanguageTool set to *picky* mode to check spelling, grammar, and punctuation on all pages of your site.

Screenshot a page

So that you learn to screenshot a whole web page, and so that our staff has a visual record of your fantastic work, use the following tool to take a single-page screenshot (saved as .png) and submit it with your .zip archive.

1. Navigate to the Log page of your website.
2. Zoom out 1 or 2 times using `cntl-` or `⌘-`.
3. Squeeze the browser window, so that it is as skinny as possible without the content spilling out of the main area.
   Take a full-page screenshot using a tool like this:
   GoFullPage: Chrome extension to take a screenshot of an entire webpage.
   FireShot: FireFox add-on to take a screenshot of an entire webpage.

4. Optimize/compress the screenshot so it is no larger than **999k**. Drag the screenshot file to the **I❤IMG Compress Image tool** to reduce the file size.
5. Rename the screenshot with your ONID userID.
6. Add the userName-screenshot.png file to the frontend folder's root (so we can find it).

What to Turn In?

- Remove your MongoDB account username and password from the backend folder's .env file.
- Zip the all the files in the **backend** folder, but *not the node_modules folder*.
- Zip the all the files in the **frontend** folder, but *not the node_modules folder*.
- Add your ONID username to each .zip archive.
- Upload both .zip archives to this Assignment.
- The grader will unzip both files, and test your site using their own local install of MongoDB.