COMP2406 Assignment 4 – Community fridge web application with MongoDB

Instructions

COMP2406 Assignment 4 – Community fridge web application with MongoDB

This assignment mostly focuses on content from Weeks 10-12 of class.

In the past three assignments, we have been building various parts of a community fridge web application. In Assignment 1, we built the client-side functionality of the application, to allow users to browse the list of fridges, browse items in a fridge, and pickup items from a fridge. In Assignment 2, we built the server-side component of the web application. Specifically, we created a Node.js server to serve all the resources associated with the application (e.g., HTML, CSS, JavaScript, CSS, and JSON). In addition to this, we also added persistence in our application. In Assignment 3, we further built the assignment in Express, specifically focusing on building a REST API.

In this assignment, we will connect our Express application to a MongoDB database. This involves moving all of the data from our JSON files into the MongoDB database and performing queries on this data.

This assignment should be completed individually. It must be completed using Node.js, Express, and MongoDB/Mongoose.

Starting code

We have provided the solution for Part 1 of assignment 3 as starting code for this assignment. If you were unable to fully complete assignment 3, you can use the provided

solution as a starting point for this assignment. We made some slight changes to the Express routes from A3, so if you would like to use your own Assignment 3, please make the required

changes. These changes are highlighted in the specification by the marker **[*****updated].**

All of the application resources should be placed on the Node.js web server. You should not save any resources on the client-side. With regards to resources, we mean all assets associated with the application such as HTML files, CSS files, JavaScript files, images, and any other such assets used by your web application.

All data associated with the application should be stored in a MongoDB database as specified in the following sections. Questions that are implemented without utilizing the database will receive a mark of 0. For example, if any of the JSON files are updated instead of the database, a mark of 0 will be given for that question. Similarly, the solution must be implemented using a REST API created in Assignment 3. This API must be updated with the necessary changes required for this assignment.

Requirements

The primary requirements of the assignment involve implementing a database for our Express application. All routes will implemented on the server-side and tested using Post-man. You do not need to implement the client-side HTML/JavaScript code associated with the routes. However, there is a bonus question, which involves implementing a client-side component of the "Search feature". As this is a bonus, it is optional for the assignment. It is meant for those who would like more practice working with the client-side code.

Part 1: Setting up a database

Begin by setting up your MongoAtlas cloud database account as explained in the lectures. We also used this account in Tutorial 9. If you did not get a chance to complete the tutorial, please watch the "Setting up a MongoDB Cloud account" video from week 10. We recommend you complete this step well in advance of completing the assignment, as not being able to setup the account will hinder your ability to successfully complete the assignment.

As shown in the lectures, your code should have a config.js file, which includes the connection information for your MongoDB database.

You must update the following information in the file to correspond to your assignment setup:

- **username:** update the value in this variable to your MongoDB username
- **password:** update the value in this variable to your MongoDB password

- **dbname:** update the dbname to community-fridge-stdnumber", where stdnumber is equal to your student number. For example, if your student number is 100978291,

then the value of the dbname variable should be "community-fridge-100978291"

Failure to update any of these values correctly will result in a mark of 0 for the assignment. The TAs need to be able to execute the code for a large number of students and completing this part incorrectly will affect their ability to mark the assignment in a timely manner.

NOTE: when you make changes to the database, you must "refresh" the MongoDB Atlas page to see the new changes. Similarly, you must restart the Node.js server anytime you make changes to your server-side code.

Part 2: Creating database schemas (6-marks)

Once the MongoDB database is setup, you need to use Mongoose to create three schemas for your database. These schemas will be used to add elements into the database. Specifically, the database should have the following three schemas:

1. **Fridges:** specifies the structure of the data associated with a fridge
2. **Items:** specifies the structure of the data associated with an Item
3. **Types:** specifies the structure of the data associated with a type

Remember, that schemas in Mongoose are used to create Models, which are used to add documents (i.e., rows) into your database. Thus, each schema will correspond to a collection in the database.

2.1: Creating a database schema for Fridges

The following are the schema requirements for a fridge and this schema should be stored in a file called fridgeModel.js

- **id:** stores a unique ID for the fridge (e.g., fg-1). The type for the id should be a String and it is a required field with a minimum length of 4 and a maximum length of 6.
- **name:** stores the name of the fridge. The type for the name should be a String. It is a required field with a reasonable minimum and maximum length. The minimum and maximums must be implemented using the built-in scheme validation in Mongoose.
- **numItemsAccepted:** indicates the number of items accepted by the fridge. It should be a number and have a default value of 0.
- **canAcceptItems:** indicates the maximum number of items that the fridge can accept. It is a required field and should have a minimum value of 1 and a maximum value of 100. The minimum and maximums must be implemented using the built-in scheme validation in Mongoose.

- [***updated] **contactInfo:** specifies the contact info for a fridge. It should be an object consisting of two properties: contactPerson and contactPhone. The type of both of

these properties should be a String. As you can see from the comm-fridge-data.json file, the structure of this field has been modified compared to previous assignments. Please update this field in your code if not using the starter-code.

- **address:** specifies the address information for a fridge. It is a required object with the following five fields{
  street: type of String
  postalCode: type of String
  city: type of String
  province: type of String,
  country: type of String
- [***updated] **acceptedTypes:** an array indicating the types of items accepted by the fridge. This field is required. The types of items accepted in previous assignments included a name (e.g., dairy, produce). We will now create a types collection, which will store all of the types available in our application. So, the acceptedTypes array now will have a list of ids instead of the names of the types (e.g., 839201, 1801920).
- **items:** an array of item objects. Each item object has two properties: id and quantity. The id of the item in the array must correspond to its id in the items collection. The quantity field should be a number.

2.2: Creating a database schema for Items

Create a schema for an Item. Specifically, the item schema should be stored in a file called itemModel.js and must have the following fields:

- **id:** stores a unique ID for the item (e.g., 1). The type for the id should be a String and it is a required field with a minimum length of 1 and a maximum length of 4.
- **name:** a required field, which stores the name of the item. It should be of type String and have a reasonable minimum and maximum length. The minimum and maximums must be implemented using the built-in scheme validation in Mongoose.
- **type:** a required field, which stores the type of the item. The type of the item should be an id which corresponds to an id in the "types" collection.
- **img:** a required field, which stores the path to an image. It should be of type String and have a reasonable minimum and maximum length. The minimum and maximums must be implemented using the built-in scheme validation in Mongoose.

2.3: Creating a database schema for Types

Create a schema for the type of an item. This schema should be stored in the a file called typeModel.js, and it should have the following field:

- **id:** stores a unique ID for the item type (e.g., 1). The id should be a String and it is a required field with a minimum length of 1 and a maximum length of 4.

- **name:** a required field, which stores the name of the type (e.g., dairy, produce). It has a type of String and a reasonable minimum and maximum length. The minimum and maximums must be implemented using the built-in scheme validation in Mongoose.

2.4: Populating the database with data from the JSON files (already completed for you)

Once the schemas are created, we will need to use them to initialize our database with contents from the comm-fridge-data.json, comm-fridge-items.json, and comm-fridge-types.json files. This part has already been completed for you.

Part 3: Creating database queries (94-marks)

Once the database is created and initialized with data from the JSON files, you will write queries for creating, retrieving, updating, and deleting data for the community fridge application. The queries must be integrated with routes in Express. Some of this will involve updating the existing Express routes created in Assignment 3, and others will involve creating new routes. Details on each are provided below.

3.1 Updating existing express routes (54-marks)

When updating the existing Express routes from Assignment 3, the goal is to modify the code so that the CRUD operations on the data are performed on the MongoDB database, instead of the underlying JSON files. None of your code should update the JSON files. If a route is completed by updating the JSON files, then you will receive 0 for that question. In addition to modifying the routes, you will also need to add additional validation on the data to ensure that it meets the schema requirements.

1. **Retrieving information for all fridges:** when a GET request with the URL /fridges is received **AND** the content-type requested by the client is JSON, then the server should perform a query on the Fridges collection of the database, and respond with the data associated with all of the fridges in the application.

2. **Retrieving data associated with a specific fridge:** when a GET request with the parameterized URL /fridges:/fridgeID is received, then the server should query the Fridges collection of the database, and respond with all of the data associated with the fridge with the fridgeID. A 404 error should be returned if the fridgeID does not exist.

3. **Adding a new fridge:** when a POST request with the URL /fridges is received, the server should add a new fridge in the Fridges collection of the database. Information

   provided in the body of the POST request should be used to perform the addition. If the body of the POST request does not meet the Fridge schema, then a 400 error

message should be returned. Otherwise, the fridge should be successfully added and 200 should be returned with information about the new fridge in the response body.

An example of a POST body which violates the Fridge schema is: {"name": "St-Laurent fridge}, because it is missing some required fields.

4. **Updating information about a fridge:** when a PUT request with the parameterized URL /fridges/:fridgeID is received, the server should update the information for the specified fridge in the Fridges collection of the database. The information to be updated should be provided in the body of the PUT request. The body of the PUT request should only contain information which was updated. The body of the PUT request should adhere to the requirements of the Fridge schema. Any violations of the schema should result in a 400 error.

   An example of a valid PUT request is: {"canAcceptItems": 50}

5. **[***updated] Adding an item in the fridge:** when a POST request with the parameterized URL /fridges/:fridgeID/items is received, the server should add an item into the specified fridge in the database. At minimum, the body of the POST request should contain information for all of the required item fields. If the item already exists in the fridge, then a 409 error message should be returned to indicate duplicates. The body of the POST request should adhere to the requirements of the Fridge schema. Any violations of the schema should result in a 400 error.

6. **[***updated] Deleting an item from a fridge:** when a DELETE request with the parameterized URL /fridges/:fridgeID/items/:itemID is received, the server should remove the specified item from the fridge in the database. In Assignment 3, this was addressed using various approaches (e.g., reducing the quantity of the item by 1 or deleting the item). When implementing this using a database, the easiest option would be to completely delete the item regardless of the quantity, which is the requirement here. Decreasing the quantity of the item would be considered an update, and should be done using a PUT request instead. We have provided a new route for it in this assignment. If you had alternate implementation for this in Assignment 3, please make the necessary adjustments to it for Assignment 4. You can review the solutions to see how to complete this route for Assignment 3. The input validation requirements for this route are the same as in Assignment 3, specifically that the fridgeID and itemID must exist before a delete can be successful. If either the fridgeID or itemID does not

   exist, a 404 error should be returned.

7. [***updated] **Delete a list of items from a fridge:** when a DELETE request with a query string is received for the URL /fridges/:fridgeID/items, the server should remove the list of items from the fridge database, as specified in the query string. For example, if the query string /fridges/:fridgeID/items?item=itemId1&item=itemId2 is received, then item1 and item2 should be removed from the fridge if they exist inside the fridge specified in the URL (e.g., fridgeID). If a query string is not provided, then **ALL OF** the items from the specified fridge should be removed. This requirement is slightly updated from Assignment 3, so please make the necessary adjustments to your code or use the provided solution. If the fridgeID or any of the itemIDs doesn't exist, then an error code of 404 should be returned with an error message.

3.2 Creating new express routes (40-marks)

In this section, you will create some new routes for the Express application to add additional functionality into the community fridge application.

1. **Updating the quantity of an item in the fridge:** when a PUT request with the parameterized URL /fridges/:fridgeID/items/:itemID is received, the server should update the item quantity for the specified fridge in the database. The updated quantity information should be provided in the body of the PUT request. Before performing the update, the route must check that the fridgeID exists, and the itemID exists inside the fridge. If these requirements are met, then the update should proceed. Upon a successful update, the server should return a status of 200 with the updated item in the response's body. If either of the IDs do not exist, then an error of 404 should be returned.

2. **Adding a new item into the items collection:** this route concerns adding to the items model/collection and **NOT** adding an item to a fridge. Thus, this route is different from Part 3.1.5. When a POST request for the route /items is received, the server should add a new item into the Items collection in the database using the information provided in the body of the POST request. The item should only be added if it is not a duplicate. Specifically, if the item has the same name as another item in the Items collection, then a 409 error should be returned. In addition, the body of the POST request must meet the Item schema. Any violations should result in a 400 error. If the item is not a duplicate and meets the Item scheme, then the item should successfully be added into the Items collection. A 200 code should be returned with the new item in the response body.

3. **Search for an item in the items collection:** when a GET request with a query string is received for the route /search/items?type=""&name="", the server should return a list

of items from the items collection which match the query parameters. For example, if the following route was received: /search/items?type=dairy&name=milk, then the server should return a list of items from the items collection that have a type of dairy and contains milk in their name. If an improperly formatted query string is received, then a 400 error should be returned.

Bonus: Client-side application (5-marks)

In this assignment, there are no requirements to connect the routes in Part 3 to a client-side application. However, we have included a bonus question for those who would like to implement a client-side component for the new search route specified in Part 3.2.3.

Specifically, as a bonus question, implement a "Search" screen which includes the following UI elements:

- **Two textboxes for the query:** one labelled "type" and the other labelled "name". These allow the user to specify the values of the type and name query parameters.
- **A submit button:** this allows the user to execute the query. This button should be labelled as "Search".

When the submit button is pressed, the values provided by the user in the textboxes should be used to form a URL which matches the route in Part 3.2.3. An AJAX request should be sent to the server to execute the search query. Results from the search, namely the item data can be displayed in the browser's console or in the "Search" screen.

Submission

Submit your code as one .zip file which contains the following:

1. A Readme text file which includes:
    a. your name
    b. your student number
    c. brief description of the contents of the .zip file
    d. any relevant instructions for running your code
2. You must also check that the config.js file is created according the specification provided.

To submit your assignment, please **zip** all of your files for the assignment and submit the zip file on Brightspace. This .zip file should be named as follows: **“COMP2406_A4_firstname.zip”**. Make sure you download the zip after submitting and verify

that the file contents are correct and your instructions are sufficient to install/run your solution.

Marking Scheme

/0 Part 1

/6 Part 2

/94 Part 3

starter_code.zip (17.87 KB)

Submissions

No submissions yet. Drag and drop to upload your assignment below.

Drop files here, or click below!

Upload Record Choose Existing

You can upload files up to a maximum of 2 GB.

Activity Details

Task: Submit to complete this assignment

Due April 12 at 11:55 PM

Last Visited Apr 5, 2022 10:33 AM