Tommy Las Z23517623 Tommy-Las

March 27th 2022 [Github Repository](https://github.com/COT4930-Spring-2022-Fullstack-Web/hw5-Tommy-Las)

Full Stack Web Development COT4930

Homework #5

Part 1 :

First, I had to install mongodb using HomeBrew in MacOS:



Then, I installed MongoDB Compass through the MongoDB website:

Graphical user interface, application, Teams

Description automatically generated

Then I needed to go through each of the examples from chapter 6 in the textbook:

createdatabase.js: It just creates a new database called music in uri “mongodb://127.0.0.1:27017/music”. You need to first nstall the mongodb package using npm install mongodb.



Createcollection.js: Creates a collection called Songs in the music database we created before. A collection is a collection of documents. A variable named “dbo is created which contains an object of the music database (dbo = client.db(“music”))” . Then it uses the dbo.createCollection method to create the Songs collection. We need to close the client object by using dbo.close()



It shows in Compass:

Graphical user interface, application

Description automatically generated

Insertdocument.js: This program adds a document to the songs collection in the music database. A document is an occurrence of a collection, which means that is data in a JSON object. It uses the insertOne method on dbo.collection object to insert the JSON object “mysong” to the songs collection. When inserted, MongoDB creates a unique id:



Graphical user interface, text, application, email

Description automatically generated

insertdocument2.js: This program is the same as above, but this one instead has a different JSON structure. It is to prove that documents does not need to have the same amount of data.



insertmany.js: This program inserts many documents into the songs collection. Inserts each song located in an array of JSON objects (documents/songs).

Text

Description automatically generated

insertmany2.js: Same as above but different data

findall.js: It displays the names of all songs in the songs collection. Uses .find() to search for all documents in the database, then uses toArray() to covert the results from .find() into an array. It loops the array and displays the name of each song.

Text

Description automatically generated

findone.js: uses the findOne() method to display the data of the first document in the songs collection.

Graphical user interface, text, application

Description automatically generated

findsome.js: It uses find() method, but in this case it uses the projection operator to specify that only the title of each song should be returned by the find method. The projection is a JSON object {-id: 0, title: 1} the 0 means it should not be returned, the 1 means it should be included.

Text

Description automatically generated

queryname.js: It uses query parameters to find a specific song in the songs collections. A query parameter needs to be placed inside find() and is a JSON object.. I had to change the code a little because it was searching for a field it did not exist. I changed the query parameter to { artist: John Lennon}. So the find method looks like find({artist: John Lennon}):



queryduration.js: Uses query parameters again to find songs with duration greater than 200 seconds. The query parameter is: { "duration": { $gt: 200 } }; the $gt operator means greater than. I had to modify the Imagine document to add the duration, I did it by using Compass.



queryand.js: Same as above, but the query parameters adds the $and operator, and the find() returns a song with duration greater than 200 and where the artist is John Lennon: { $and: [ {"artist": "John Lennon"} , {"duration": { $gt: 200 }} ] }; with the $and operator, you need to have the conditions in an array. I had to modify the conditions from query from the example of the book because it was not finding anything.

Graphical user interface, text, application

Description automatically generated

queryregex.js: Uses a regular expression as query in order to find songs that start with the letter “I”, the query : { "title": /^I/ }



queryregex2.js: Uses a regular expression as query in order to find songs that end with the letter “n” (book example is with s, but I don’t have any songs in the database with that requirement), the query ends up being : { "title": /$n/ }

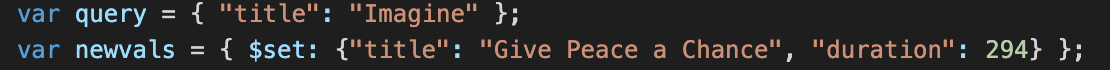


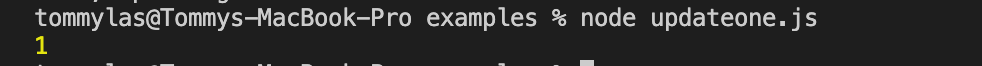
findsort.js: Displays the duration and title of the songs sorted by duration. Uses find().sort(mysort) where my sort has the criteria in a JSON object: {“duration”: 1}. The 1 means ascending order.

Graphical user interface, text, application

Description automatically generated

updateone.js: Updates the data of a specific song. The song with title Imagine will be changed to Give Peace a Chance, and the duration is changed to 294. It uses the updateOne() method, with parameters:



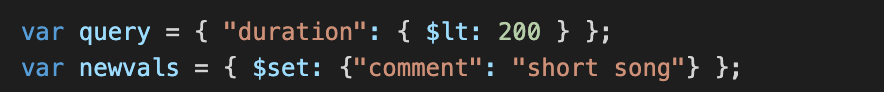


In compass shows it was updated:

A picture containing graphical user interface

Description automatically generated

updatemany.js: Songs with duration less than 200 will be added a field comment: “short song”. This program instead uses the updateMany() method, will update songs based on the query parameters. The query and the new values that will be included in the updateMany() method.





Graphical user interface, text, application, email

Description automatically generated

deleteone.js: Deletes a document in the songs collection in the music database. Uses the deleteOne(query) operator and the query includes parameters for the document that wants to be deleted. In this case we want the song with title “Imagine” to be deleted from the songs collection and music database.



Text

Description automatically generated

deletemany.js: Deletes songs where the title ends with the letter s. In this case the query parameters would look like this: 

Text

Description automatically generated with medium confidence

deletecollection.js: Deletes the songs collection from the music database. Uses dbo.collection(“songs”).drop() method to delete it.



deletedb.js : uses de dropDatabase() to delete the music database



Database no longer exists:

Graphical user interface, application

Description automatically generated

Part 2:

For the second part of this assignment I had to implement the same program as HW4, but instead of storing the students in JSON files in the server, we store data in the MongoDB database.

I created a collection in postman to save and use all requests:

Graphical user interface, application

Description automatically generated

Connecting to the database is simple, you need to first install the mongodb package using npm install mongodb, and then add it to your code using require(‘mongodb’).client. Then you do:



This connects you to the database, and then you can do any mongodb method to access the database. For the POST request, I need to check for duplicates first, and if no user has the same name and last name, add a student to the database. The algorithm for checking for duplicates

first uses the find() method to find a user with the first and last name that the user input. If the response of the method returns an array with elements, it means that the user is already in the database, so you send a error message. If the returned array is empty, it means that we can add the student to the database. I used the insertOne mongodb method to add a student to the students collection in the hw5 database. I did not include any code to create the hw5 database or the students collection. This is because when you add a student and the database or the collection does not exist, mongo creates it for you. Here is the postman POST request:

Graphical user interface, text, application, email

Description automatically generated

Duplicate based on first & last name:

Graphical user interface, text, application, email

Description automatically generated

Proof that it was added to the the database, I added more students:

Graphical user interface, text, application, email

Description automatically generated

For the GET request for /students, first I check if there any query parameters. If there are no query parameters, it means that it was a GET request to display all students in the students collection. If the query parameters object is not empty, it means that the user wants to search for a student with their first name and last name. If the query parameters is not empty, I run the method:



Where the query contains the name and last name of the user we need to find. It sends an error message with status 404 if the user is not found, if the user is found, it sends the student information as JSON object with status 200.

If the query parameters is empty, then it runs the .find method which will return an array of json objects, which are all students in the students collection. It sends that with a status 200.

ALL students:

Graphical user interface, text, application, email

Description automatically generated

GET student based on first name and last name:

Graphical user interface, application, email

Description automatically generated

To get the information of a student based on the record\_id, in this case it uses /students/:record\_id. I connect to the database and used the find(query) method from mongodb, where query contains the record\_id of the user that need to be find. If the find method returns an empty array, it will send an error message that the user was not found. If it returns a non empty array, it means that the user was found and it sends it as an JSON object.

Graphical user interface, text, application, email

Description automatically generated

To replace a student information, I send a PUT request to /students/:record\_id. The user inputs the \_id of the student it wants to update and new values. I use updateOne() method:



Where the query contains the id of the student it needs to be replaced, and newvals contains the new values of that user. If the user is found and updated, it sends a success message. To check whether the user was modified or not, the updateOne method returns an object where one element is “modifiedCount”. If the value is 0 it means that no user was updated, maybe because the id given is incorrect, so I send an error message. If modifiedCount is 1 , it means that one student was modified so I send a success message.

Graphical user interface, text, application, email

Description automatically generated

Show in database:

Graphical user interface, text, application, email

Description automatically generated

To delete a student, I send a DELETE request to /students/:record\_id. It uses de deleteOne() method from mongodb to delete the student: 

Where query contains the record id of the user it was to delete. The deleteOne method returns a response object where one element is “deletedCount”. If the value is 0, it means that no student was deleted and I send an error message. If the value is one it means that it was deleted and I send a success message.

Graphical user interface, text, application, email

Description automatically generated

Show in database:

Graphical user interface, text, application, email

Description automatically generated

Before sending back a status and a message/data, I need to close the connection to the mongoDB database using client.close().

Part 3:

I used the same front end from HW4, but I added the GET request to display the information of all students, which I missed on HW4. I created a table and hid it using some css. When the button is clicked, the table is displayed and dynamically adds rows to the table to display each student information. This post from stackoverflow helped me with this:

<https://stackoverflow.com/questions/171027/add-table-row-in-jquery>

Graphical user interface

Description automatically generated