Lab 4

New Attempt

Due Feb 28 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip

CS-546 Lab 4

MongoDB

For this lab, we are going to make a few parts of a band database. You will create the first of these data modules, the module to handle a listing of bands.

You will:

- Separate concerns into different modules.
- Store the database connection in one module.
- Define and store the database collections in another module.
- Define your Data manipulation functions in another module.
- Continue practicing the usage of `async` / `await` for asynchronous code
- Continuing our exercises of linking these modules together as needed

Packages you will use:

You will use the **mongodb (https://mongodb.github.io/node-mongodb-native/)** package to hook into MongoDB

You **may** use the **lecture 4 code (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_04)** as a guide.

You must save all dependencies you use to your package.json file

How to handle bad data

```
async function divideAsync(numerator, denominator) {
    if (typeof numerator !== "number") throw new Error("Numerator needs to be a number");
    if (typeof denominator !== "number") throw new Error("Denominator needs to be a number");

    if (denominator === 0) throw new Error("Cannot divide by 0!");

    return numerator / denominator;
}

async function main() {
    const six = await divideAsync(12, 2);
    console.log(six);

    try {
        const getAnError = await divideAsync("foo", 2);
    } catch(e) {
```

```
        console.log("Got an error!");
        console.log(e);
    }

}

main();
```

Would log:

```
6
"Got an error!"
"Numerator needs to be a number"
```

Folder Structure

You will use the folder structure in the stub. for the data module and other project files. You can use mongoConnection.js, mongoCollections.js and settings.json from the lecture code, however, you will need to modify settings.json and mongoCollections.js to meet this assignment's requirements. There is an extra file in the stub called helpers.js. You can add all your helper/validation functions in that file to use in your other modules.

YOU MUST use the directory and file structure in the code stub or points will be deducted. You can download the starter template here: **lab4_stub.zip (https://sit.instructure.com/courses/64568/files/11206064?wrap=1)** ⬇ **(https://sit.instructure.com/courses/64568/files/11206064/download?download_frd=1)** **PLEASE NOTE: THE STUB DOES NOT INCLUDE THE PACKAGE.JSON FILE. YOU WILL NEED TO CREATE IT! DO NOT FORGET TO ADD THE START COMMAND AND "type": "module". DO NOT ADD ANY OTHER FILE OR FOLDER APART FROM PACKAGE.JSON FILE.**

Database Structure

You will use a database with the following structure:

- The database will be called **FirstName_LastName_lab4**
- The collection you use will be called `bands`

bands

The schema for a band is as followed:

```
{
    _id: ObjectId,
    name: string,
    genre: [strings],
    website: string (must contain full url http://www.patrickseats.com),
    recordCompany: string,
    groupMembers: [strings],
    yearBandWasFormed: number,

}
```

The `_id` field will be automatically generated by MongoDB when a band is inserted, so you do not need to provide it when a band is created.

An example of how Pink Floyd would be stored in the DB:

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "Pink Floyd",
    genre: ["Progressive Rock", "Psychedelic rock", "Classic Rock"],
    website: "http://www.pinkfloyd.com",
    recordCompany: "EMI",
    groupMembers: ["Roger Waters", "David Gilmour", "Nick Mason", "Richard Wright", "Sid Barrett" ],
    yearBandWasFormed: 1965
}
```

bands.js

In bands.js, you will create and export five methods:

Remember, you must export methods named precisely as specified. The async keyword denotes that this is an async method.

async create(name, genre, website, recordCompany, groupMembers, yearBandWasFormed);

This async function will return to the newly created band object, with **all** of the properties listed above.

If `name, genre, website, recordCompany, groupMembers, yearBandWasFormed` are not provided at all, the method should throw. (**All fields need to have valid values**);

If `name, website, recordCompany` are not `strings` or are empty strings, the method should throw.

If `website` does not contain `http://www.` and end in a `.com`, and have at least 5 characters in-between the `http://www.` and `.com` this method will throw.

If `genre, groupMembers` are not arrays and if they do not have **at least** one element in each of them that is a valid `string`, or are empty strings the method should throw. (each element should be a valid string but the arrays should contain at LEAST one element that's a valid string.

If `yearBandWasFormed` is not a `number`, or if `yearFormed` is less than 1900 or greater than the current year (2023) the the method should throw. (so only years 1900-2023 are valid values)

Note: FOR ALL INPUTS: Strings with empty spaces are NOT valid strings. So no cases of " " are valid.

For example:

```
import * as bands from "./bands.js";

async function main() {
    const pinkFloyd = await bands.create("Pink Floyd", ["Progressive Rock", "Psychedelic rock", "Classic
Rock"], "http://www.pinkfloyd.com", "EMI", ["Roger Waters", "David Gilmour", "Nick Mason", "Richard Wrigh
t", "Sid Barrett" ], 1965);
```

```
    console.log(pinkFloyd);
}

main();
```

Would return and log:

```
{
    _id: "507f1f77bcf86cd799439011",
    name: "Pink Floyd",
    genre: ["Progressive Rock", "Psychedelic rock", "Classic Rock"],
    website: "http://www.pinkfloyd.com",
    recordCompany: "EMI",
    groupMembers: ["Roger Waters", "David Gilmour", "Nick Mason", "Richard Wright", "Sid Barrett" ],
    yearBandWasFormed: 1965
}
```

This band will be stored in the **bands** collection.

If the band cannot be created, the method should throw.

Notice the output does not have ObjectId() around the ID field and no quotes around the key names, you need to display it as such.

async getAll();

This function will return an array of all bands in the collection. **If there are no bands in your DB, this function will return an empty array**

```
import * as bands import "./bands.js";

async function main() {
    const allBands = await bands.getAll();
    console.log(allBands);
}

main();
```

Would return and log all the bands in the database.

```
[{
    _id: "507f1f77bcf86cd799439011",
    name: "Pink Floyd",
    genre: ["Progressive Rock", "Psychedelic rock", "Classic Rock"],
    website: "http://www.pinkfloyd.com",
    recordCompany: "EMI",
    groupMembers: ["Roger Waters", "David Gilmour", "Nick Mason", "Richard Wright", "Sid Barrett" ],
    yearBandWasFormed: 1965
},
{
    _id: "507f1f77bcf86cd799439012",
    name: "The Beatles",
    genre: ["Rock", "Pop", "Psychedelia"],
    website: "http://www.thebeatles.com",
    recordCompany: "Parlophone",
    groupMembers: ["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"],
    yearBandWasFormed: 1960
},
{
    _id: "507f1f77bcf86cd799439013",
    name: "Linkin Park",
    genre: ["Alternative Rock", "Pop Rock", "Alternative Metal"],
    website: "http://www.linkinpark.com",
```

```
    recordCompany: "Warner",
    groupMembers: ["Chester Bennington", "Rob Bourdon", "Brad Delson", "Mike Shinoda", "Dave Farrell", "J
oe Hahn"],
    yearBandWasFormed: 1996
}
]
```

Notice the output does not have ObjectId() around the ID field and no quotes around the key names, you need to display it as such.

async get(id);

When given an id, this function will return a band from the database.

If no `id` is provided, the method should throw.

If the `id` provided is not a `string`, or is an empty string, the method should throw.

If the `id` provided is not a valid `ObjectId`, the method should throw

If the no band exists with that `id`, the method should throw.

For example, you would use this method as:

```
import * as bands from "./bands.js";

async function main() {
    const linkinPark = await bands.get("507f1f77bcf86cd799439013");
 console.log(linkinPark);
}

main();
```

Would return and log Linkin Park:

```
{
    _id: "507f1f77bcf86cd799439013",
    name: "Linkin Park",
    genre: ["Alternative Rock", "Pop Rock", "Alternative Metal"],
    website: "http://www.linkinpark.com",
    recordCompany: "Warner",
    groupMembers: ["Chester Bennington", "Rob Bourdon", "Brad Delson", "Mike Shinoda", "Dave Farrell", "J
oe Hahn"],
    yearBandWasFormed: 1996
}
```

Notice the output does not have ObjectId() around the ID field and no quotes around the key names, you need to display it as such.

Important note: The ID field that MongoDB generates is an `ObjectId`. This function takes in a `string` representation of an ID as the `id` parameter. You will need to convert it to an `ObjectId` in your function before you query the DB for the provided ID and then pass that converted value to your query.

```
//We need to require ObjectId from mongo
let { ObjectId } = require('mongodb');
```

```
/*For demo purposes, I will create a new object ID and convert it to a string,
Then will pass that valid object ID string value into my function to check if it's a valid Object ID (whi
ch it is in this case)
I also pass in an invalid object ID when I call my function to show the error */


let newObjId = new ObjectId(); //creates a new object ID


let x = newObjId.toString(); // converts the Object ID to string


console.log(typeof x); //just logging x to see it's now type string


//The below function takes in a string value and then attempts to convert it to an ObjectId


function myDBfunction(id) {


  //check to make sure we have input at all
  if (!id) throw 'Id parameter must be supplied';


  //check to make sure it's a string
  if (typeof id !== 'string') throw "Id must be a string";


  //Now we check if it's a valid ObjectId so we attempt to convert a value to a valid object ID,
  //if it fails, you will throw an error
  if (!ObjectId.isValid(id) throw "ID is not a valid Object ID";

  console.log('Valid Object ID, now I can pass ObjectId(id) as the ID into my query.');
}

//passing a valid string that can convert to an Object ID
try {
  myDBfunction(x);
} catch (e) {
  console.log(e.message);
}

//passing an invalid string that can't be converted into an object ID:
try {
  myDBfunction('test');
} catch (e) {
  console.log(e.message);
}
```

async remove(id)

This function will remove the band from the database.

If no `id` is provided, the method should throw.

If the `id` provided is not a `string`, or is an empty string the method should throw.

If the `id` provided is not a valid `ObjectId`, the method should throw

If the band cannot be removed (does not exist), the method should throw.

If the removal succeeds, return the name of the band and the text " has been successfully deleted!"

```
import * as bands from "./bands.js";

async function main() {
```

```
    const removeBeatles = await bands.remove("507f1f77bcf86cd799439012");
 console.log(removeBeatles);
}
main();
```

Would return and then log: "The Beatles has been successfully deleted!". **NOTE: YOU MUST RETURN THIS EXACT SENTENCE. Points will be deducted if you do not match this format 100% including the "!" and including the band name that was deleted. It must match 100% or points will be deducted.**

Important note: The ID field that MongoDB generates is an `ObjectId`. This function takes in a `string` representation of an ID as the `id` parameter. You will need to convert it to an `ObjectId` in your function before you query the DB for the provided ID. See example above in `getId()`.

async rename(id, newName)

This function will update the name of the band currently in the database.

If no `id` is provided, the method should throw.

If the `id` provided is not a `string`, or is an empty string the method should throw.

If the `id` provided is not a valid `ObjectId`, the method should throw.

If `newName` is not provided, the method should throw.

If `newName` is not a `string`, or an empty string, the method should throw.

If the band cannot be updated (does not exist), the method should throw.

if the `newName` is the same as the current value stored in the database, the method should throw.

If the update succeeds, return the entire band object as it is after it is updated.

```
import * as bands from "./bands.js";

async function main() {
    const renamedBeatles = await bands.rename("507f1f77bcf86cd799439012", "Lennon's Boys");
 console.log(renamedBeatles);
}
main();
```

Would log the updated band:

```
{
    _id: "507f1f77bcf86cd799439012",
    name: "Lennon's Boys",
    genre: ["Rock", "Pop", "Psychedelia"],
    website: "http://www.thebeatles.com",
    recordCompany: "Parlophone",
    groupMembers: ["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"],
    yearBandWasFormed: 1960
}
```

Notice the output does not have ObjectId() around the ID field and no quotes around the key names, you need to display it as such.

Important note: The ID field that MongoDB generates is an `ObjectId`. This function takes in a `string` representation of an ID as the `id` parameter. You will need to convert it to an `ObjectId` in your function before you query the DB for the provided ID. See example above in `getId()`.

app.js

For your app.js file, you will:

1. Create a band of your choice.
2. Log the newly created band. (Just that band, not all bands)
3. Create another band of your choice.
4. Query all bands, and log them all
5. Create the 3rd band of your choice.
6. Log the newly created 3rd band. (Just that band, not all bands)
7. Rename the first band
8. Log the first band with the updated name.
9. Remove the second band you created.
10. Query all bands, and log them all
11. Try to create a band with bad input parameters to make sure it throws errors.
12. Try to remove a band that does not exist to make sure it throws errors.
13. Try to rename a band that does not exist to make sure it throws errors.
14. Try to rename a band passing in invalid data for the `newName` parameter to make sure it throws errors.
15. Try getting a band by ID that does not exist to make sure it throws errors.

General Requirements

1. You **must not submit** your node_modules folder or package-lock.json
2. You **must remember** to save your dependencies to your package.json folder
3. You must do basic error checking in each function
4. Check for arguments existing and of proper type.
5. Throw if anything is out of bounds (ie, trying to perform an incalculable math operation or accessing data that does not exist)
6. If a function should return a promise, you should mark the method as an `async` function and return the value. Any promises you use inside of that, you should *await* to get their result values. If the promise should reject, then you should throw inside of that promise in order to return a rejected promise automatically. Thrown exceptions will bubble up from any awaited call that throws as well, unless they are caught in the async method.
7. You **must remember** to update your package.json file to set `app.js` as your starting script!
8. You **must** submit a zip file named in the following format: `LastName_FirstName_CS546_SECTION.zip (ie. Hill_Patrick_CS546_WS.zip)`

Lab 4 Rubric

Criteria	Ratings		Pts
bands.js - create Test cases used for grading will be different from assignment examples.	**40 to >0.0 pts** **2 Points/Error Handling Test Case & 5 Points/Valid Input Test Case** 15 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, returns correct data, and few or all test cases pass.	**0 pts** **All Test Cases Failed** Incorrect implementation, none of the test cases pass, or the function does not return anything.	40 pts
bands.js - getAll Test cases used for grading will be different from assignment examples.	**10 to >0.0 pts** **10 Points/Valid Input Test Case** 1 Valid Input Test Cases. The function is implemented correctly, returns correct data, and all test cases pass.	**0 pts** **All Test Cases Failed** Incorrect implementation, none of the test cases pass, or the function does not return anything.	10 pts
bands.js - get Test cases used for grading will be different from assignment examples.	**15 to >0.0 pts** **1 Points/Error Handling Test Case & 10 Points/Valid Input Test Case** 5 Error Handling Test Cases & 1 Valid Input Test Cases. The function is implemented correctly, returns correct data, and few or all test cases pass.	**0 pts** **All Test Cases Failed** Incorrect implementation, none of the test cases pass, or the function does not return anything.	15 pts
bands.js - remove Test cases used for grading will be different from assignment examples.	**15 to >0.0 pts** **1 Points/Error Handling Test Case & 10 Points/Valid Input Test Case** 5 Error Handling Test Cases & 1 Valid Input Test Cases. The function is implemented correctly, returns correct data, and few or all test cases pass.	**0 pts** **All Test Cases Failed** Incorrect implementation, none of the test cases pass, or the function does not return anything.	15 pts
bands.js - rename Test cases used for grading will be different from assignment examples.	**20 to >0.0 pts** **2 Points/Error Handling Test Case & 5 Points/Valid Input Test Case** 5 Error Handling Test Cases & 2 Valid Input Test Cases. The function is implemented correctly, returns correct data, and few or all test cases pass.	**0 pts** **All Test Cases Failed** Incorrect implementation, none of the test cases pass, or the function does not return anything.	20 pts
			Total Points: 100