CS-546 Lab 6

Band Application:

For this lab, we are going to make **more** parts of the database for a bands, as well as developing a simple API to interact with your application.

We will be practicing:

* Seperating concerns into different modules:
* Database connection in one module
* Collections defined in another
* Data manipulation in another
* Practicing the usage of **async / await** for asynchronous code
* Continuing our exercises of linking these modules together as needed
* Developing a simple (10 route) API server

Packages you will use:

You will use the [mongodb (Links to an external site.)](https://mongodb.github.io/node-mongodb-native/) package to hook into MongoDB

You may use the [lecture 4 code (Links to an external site.)](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_04/code) and the [lecture 5 code (Links to an external site.)](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_05/code) and [lecture 6 code (Links to an external site.)](https://github.com/Stevens-CS546/CS-546/tree/master/Lecture%20Code/lecture_07) as a guide.

You can read up on [express (Links to an external site.)](http://expressjs.com/) on its home page. Specifically, you may find the [API Guide section on requests (Links to an external site.)](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_06/code) useful.

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

Folder Structure

You will use the following folder structure for the data module. **You may need other files to handle the connection to the database as well.**

./

../data/

../data/bands.js

../data/index.js

../data/albums.js

../routes/

../routes/bands.js

../routes/index.js

../routes/albums.js

../app.js

../package.json

I also recommend having your database settings centralized in files, such as:

./

../config/

../config/mongoConnection.js

../config/mongoCollections.js

Database Structure

You will use a database with the following structure:

* The database will be called **FirstName\_LastName\_lab6**
* The collection you use to store bands will be called bands
* The collection you use to store albums will be called albums

bands

The schema for bands is now as followed:

{

"\_id": "", //STRING OR OBJECT ID

"bandName": "", //STRING

"bandMembers": [], //ARRAY OF STRINGS   
 "yearFormed": , //NUMBER  
 "genres": [], //ARRAY OF STRINGS  
 "recordLabel": "", //STRING,  
 "albums": [] //ARRAY OF ALBUM IDs as STRINGS

}

The \*\*\_id\*\* field will be automatically generated by MongoDB, so you do not need to provide it.

An example of this, for Pink Floyd

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd",

"bandMembers": ["Roger Waters","David Mason"],

"yearFormed": 1965,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "EMI",  
 "albums": ["a4f8512b9a734baf863ff33ffbabab2d"]

}

albums

The schema for albums is now as followed:

{

"\_id": "", //STRING OR OBJECT ID

"title": "", // String title

"author": "", // ID of band as string

"songs": [] // Array of strings

}

The \*\*\_id\*\* field will be automatically generated by MongoDB, so you do not need to provide it.

An example of this, for an album for the band Pink Floyd:

{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author": "507f1f77bcf86cd799439011",

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall Part 1"]   
}

data/bands.js

In bands.js, you will create and export the same methods as the in-class exercise. **You will need to modify the function to add a band to the DB to include an empty array of albums.  When you compose the band object to pass into your Mongo insert function, just set albums: []. Meanwhile, the array of albums should just contain the corresponding ids.**

data/albums.js

You will create and export the required methods to perform Create, Read, Update, and Delete methods on the albums structures. **The function you create to create an album must also modify the band object and insert the album information into the album array in the band object**

routes/bands.js

GET /bands

Getting this route will return a list of all bands in the system.

When printing a band over a route, the band will be in the following structure:

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd",

"bandMembers": ["Roger Waters","David Mason"],

"yearFormed": 1965,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "EMI",  
 "albums": [ {

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author": "507f1f77bcf86cd799439011",

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
 } ]  
}

Rather than showing just the \_id of the album, you will show the \_id, title, author, and songs for each album.

POST /bands

You should expect the following JSON to be submitted:

{

"bandName": "Da Band",

"bandMembers": ["Bob Hurt", "Angie Leslie"],  
 "yearFormed": 1999,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "Columbia Records"  
 }

If the JSON provided does not match that schema, you will issue a 400 status code and end the request. In your DB function where you create the band object to pass into the insert, initialize albums to be an empty array

If the JSON is valid and the band can be created successfully, you will return the newly created band with a 200 status code. **Remember, when accessing bands via the API, their albums are included, so when you first create the band, there will be no albums tied to them so you would just return the empty array from the DB like so:**

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd",

"bandMembers": ["Roger Waters","David Mason"],

"yearFormed": 1965,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "EMI",  
 "albums": []  
}

GET /bands/{id}

Example: GET /bands/507f1f77bcf86cd799439011

If no band that \_id is found, you will issue a 404 status code and end the request.

You will return the band with a 200 status code. **Remember, when accessing bands via the API, their albums are included if there are any albums for that band**

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd",

"bandMembers": ["Roger Waters","David Mason"],

"yearFormed": 1965,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "EMI",  
 "albums": [{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author": "507f1f77bcf86cd799439011",

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
 }  
}

PUT /bands/{id}

Example: PUT /bands/507f1f77bcf86cd799439011

This request will update band with information provided from the PUT body.

You should expect the following JSON to be submitted:

{

"bandName": "Da Band",

"bandMembers": ["Bob Hurt", "Angie Leslie"],  
 "yearFormed": 1999,  
 "genres": ["Rock", "Pop"],  
 "recordLabel": "Columbia Records"  
 }

**Remember, in a PUT request ALL fields of the band object need to be supplied.  If that band has albums, you will leave those intact and not update them. (you can read them from the original data and then in your update function include the album data in the update**

If the JSON provided in the PUT body is not as stated above, fail the request with a 400 error and end the request.

If no band exist with an \_id of {id}, return a 404 and end the request.

DELETE /bands/{id}

Example: DELETE /bands/507f1f77bcf86cd799439011

If no band exist with an \_id of {id}, return a 404 and end the request.

This route will DELETE the band with the provided id **and all albums written by this band**. Upon deletion, you will return the following structure:

Note the key "deleted" which is set to true

{

"deleted": true,

"data": {

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd",

"bandMembers": ["Roger Waters","David Mason"],

"yearFormed": 1965,  
 "genres":["Rock", "Pop"],  
 "recordLabel": "EMI",  
 "albums": [ {

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author": "507f1f77bcf86cd799439011",

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
 }  
 ]  
}  
}

routes/albums.js

GET /albums

Getting this route will return a list of all albums in the system

When printing an album over a route, the album will be in the following structure:

{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",  
 "author":

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd"

}

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
}

Along with the \_id of the author, you will also show the bandName of the author

POST /albums

You should expect the following JSON to be submitted:

{      
   "title": "", // String title

"author": "", // STRING OR OBJECT ID

"songs": [] // Array of strings

}

If the JSON provided does not match that schema, you will issue a 400 status code and end the request.

If the JSON is valid and the album can be created successful, you will return the newly created album with a 200 status code. **Remember, when accessing albums via the API, their author's band name is included as well**

{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",  
 "author":

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd"

}

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
}

GET /albums/{id}

Example: GET /albums/a4f8512b9a734baf863ff33ffbabab2d

If no album with that \_id is found, you will issue a 404 status code and end the request.

You will return the album with a 200 status code. **Remember, when accessing albums via the API, their author's band name is include as well.**

{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",  
 "author":

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd"

}

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall"]   
}

PATCH /albums/{id}

Example: PATCH /albums/a4f8512b9a734baf863ff33ffbabab2d

This request will update an album with information provided from the PATCH body.

You should expect the following JSON to be submitted:

{

"newTitle": "new title",

"newSongs: "songs to add to the album"

}

For example:

{

"newSongs: "Mother"

}

Would result in:

{

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author":

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd"   
}  
 "songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall", "Mother"]   
}

**Only one of those fields is required; you may submit this request with either newTitle, newSongs, or both!**

If the JSON provided in the PATCH body is not as stated above, fail the request with a 400 error and end the request.

If no album exists with an \_id of {id}, return a 404 and end the request.

DELETE /albums/{id}

Example: DELETE /albums/a4f8512b9a734baf863ff33ffbabab2d

If no album exists with an \_id of {id}, return a 404 and end the request.

This route will DELETE the album with the provided id. Upon deletion, you will return the following structure:

Note the key "deleted" which is set to true

{

"deleted": true,

"data": {

"\_id": "a4f8512b9a734baf863ff33ffbabab2d",

"title": "The Wall",

"author":

{

"\_id": "507f1f77bcf86cd799439011",

"bandName": "Pink Floyd"

}

"songs": ["In the Flesh?", "The Thin Ice", "Another Brick in the Wall", "The Mother"]   
 }   
}

**This route should also remove the album from the band object**

app.js

Your index.js file will start the express server on **port 3000**, and will print a message to the terminal once the server is started.

General Requirements

1. You **must not submit** your node\_modules folder
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 throw, 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 will zip up all folders and files shown in the "Folder Structure" of this assignment above
8. You **must remember** to update your package.json file to set app.js as your starting script!
9. You **must** submit a zip file named in the following format: LastName\_FirstName\_CS546\_SECTION.zip