Lab 6

Due Oct 24, 2021 by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip

CS-546 Lab 6

A Restaurant API

For this lab, you will create a simple server that provides an API for someone to Create, Read, Update, and Delete restaurants and also restaurant reviews.

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 (9 route) API server

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/code)** and the **lecture 5 code (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_05/code)** and **lecture 6 code (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_06/code)** as a guide.

You can read up on **express (http://expressjs.com/)** on its home page. Specifically, you may find the **API Guide section on requests (http://expressjs.com/en/4x/api.html#req)** 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/restaurants.js
../data/index.js
../data/reviews.js
../routes/
../routes/restaurants.js
../routes/index.js
../routes/reviews.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 restaurants will be called `restaurants` you will store a sub-document of `reviews`

restaurants

The schema for restaurants is now as followed:

```
{
    _id: ObjectId,
    name: string,
    location: string,
    phoneNumber: string (xxx-xxx-xxxx format),
    website: string (must contain full url http://www.patrickseats.com),
    priceRange: string (from $ to $$$$),
    cuisines: [strings],
    overallRating: number (from 0 to 5 this will be a computed average from all the reviews posted for a rest
aurant,
    the initial value of this field will be 0 when a restaurant is created),
    serviceOptions: {dineIn: Boolean, takeOut: Boolean, delivery: Boolean},
    reviews: [] (an array of review objects, you will initialize this field to be an empty array when a resta
urant is created)
}
```

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

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

An example of how The Saffron Lounge would be stored in the DB when it's first created:

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "The Saffron Lounge",
    location: "New York City, New York",
    phoneNumber: "123-456-7890",
    website: "http://www.saffronlounge.com",
    priceRange: "$$$$",
    cuisines: ["Cuban", "Italian" ],
    overallRating: 0
    serviceOptions: {dineIn: true, takeOut: true, delivery: false},
    reviews: []
}
```

The Restaurant Review Sub-document (stored within the restaurants document)

```
{
  _id: ObjectId,
  title: string,
  reviewer: string,
  rating: number (from 1 to 5),
  dateOfReview: string (must be a valid date string),
  review: string
}
```

For example a review:

```
{
  _id: ObjectId("603d992b919a503b9afb856e"),
  title: "This place was great!",
  reviewer: "scaredycat",
  rating: 5,
  dateOfReview: "10/13/2021",
  review: "This place was great! the staff is top notch and the food was delicious!  They really know how to
 treat their customers"
}
```

NOTE: When a review is added, you must calculate/re-calculate the average of all reviews and update the `overallRating` field in the main document to be the average of all the review ratings.

data/restaurants.js

In restaurants, you will create and export 5 methods. Create, Read (one for getting all and also one getting by id), Update, and Delete. **You must do FULL error handling and input checking for ALL functions as you have in previous labs, checking if input is supplied, correct type, range etc. and throwing errors when you encounter bad input. You can use the functions you used for lab 4 however, you will need to create an update function. rename from lab 4 will not be used.**

As a reminder, you will keep the same function names you used in lab 4:

`create(name, location, phoneNumber, website, priceRange, cuisines, serviceOptions) Note: notice we no longer pass overallRating into the function since that will be a computed field` `getAll()`

`get(id)`

`remove(id)`

`update (id, name, location, phoneNumber, website, priceRange, cuisines, serviceOptions)  Note: this is the new function you will create`

For the functions you did in lab 4, all the same input requirements apply in this lab, for new update function, those requirements are stated below.

NOTE: For create: We no longer will need to pass in `overallRating` like we did in lab 4 since that will now be a computed average field of all ratings. When a restaurant is created, in your DB function, you will initialize the reviews array to be an empty array (since there can't be reviews of a restaurant until the restaurant is in the system). You will also initialize `overallRating` to be 0 when a restaurant is created.

async update(id, name, location, phoneNumber, website, priceRange, cuisines, serviceOptions)

This function will update **all** the data of the restaurant currently in the database.

If `id, name, location, phoneNumber, website, priceRange, cuisines, serviceOptions` are not provided at all, the method should throw. (**All fields need to have valid values**);

If `id, name, location, phoneNumber, website, priceRange` are not `strings` or are empty strings, the method should throw.

If `phoneNumber` does not follow this format: xxx-xxx-xxxx, the method will 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 `priceRange` is not between '`$`' to '`$$$$`', this method will throw.

If `cuisines` is not an array and if it does not have at least one element in it that is a valid `string`, or are empty strings the method should throw.

If `serviceOptions` is not an `object`, the method should throw.

If `serviceOptions.dineIn, serviceOptions.takeOut, serviceOptions.delivery` are not in the object, the method should throw.

If `serviceOptions.dineIn, serviceOptions.takeOut, serviceOptions.delivery` is not a boolean, the method should throw.

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

data/reviews.js

In reviews.js, you will create and export 4 methods. Create, Read (one for getting all and also one getting by id), and Delete. Y**ou must do FULL error handling and input checking for ALL functions as you have in previous labs, checking if input is supplied, correct type, range etc. and throwing errors when you encounter bad input.**

Function Names:

`create(restaurantId, title, reviewer, rating, dateOfReview, review)`

`getAll(restaurantId)`

`get(reviewId)`

`remove(reviewId)`

async create(restaurantId, title, reviewer, rating, dateOfReview, review);

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

If `restaurantId, title, reviewer, rating, dateOfReview, review` are not provided at all, the method should throw. (**All fields need to have valid values**);

If `restaurantId, title, reviewer, dateOfReview, review` are not `strings` or are empty strings, the method should throw.

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

If the restaurant doesn't exists with that `restaurantId`, the method should throw

If `rating` is not a number that's value is 1-5, the method will throw.

If `dateOfReview` is not a valid date string, the method will throw.

if `dateOfReview` is prior or after the current day's date, the method should throw. (you can't leave a review yesterday or before yesterday, and you can't leave a review in the future)

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

NOTE: When a review is added, you must calculate/re-calculate the average of all reviews and update the `overallRating` field in the main document to be the average of all the review ratings.

async getAll(restaurantId);

This function will return an array of objects of the reviews given the `restaurantId`. **If there are no reviews for the restaurant, this function will return an empty array**

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

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

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

If the restaurant doesn't exists with that `restaurantId`, the method should throw.

async get(reviewId);

When given a `reviewId`, this function will return a review from the restaurant.

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

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

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

If the review doesn't exists with that `reviewId`, the method should throw.

async remove(reviewId):

This function will remove the review from the restaurant in the database.

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

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

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

If the review doesn't exists with that `reviewId`, the method should throw.

General note on all data functions: Whenever you return data that has an _id included in the returned data, return it as a string as shown in all the examples below. Do NOT return it as an ObjectId

NOTE: When a review is removed, you must calculate/re-calculate the average of all reviews and update the `overallRating` field in the main document to be the average of all the review ratings.

routes/restaurants.js

You must do FULL error handling and input checking for ALL routes! checking if input is supplied, correct type, range etc. and responding with proper status codes when you encounter bad input. All the input types, values and ranges that apply to the DB functions apply to the

routes as well so you will do all the same checks in the routes that you do in the DB functions before sending the data to the DB function

`GET /restaurants`

Responds with an array of all restaurants in the format of `{"_id": "restaurant_id", "name": "restaurant_name"}` Note: Notice you are **ONLY** returning the restaurant ID as a **string**, and restaurant name

```
[{ "_id": "603d965568567f396ca44a72","name": "The Saffron Lounge"},{ "_id": "704f456673467g306fc44c34","name
": "Black Bear"},.....]
```

`POST /restaurants`

Creates a restaurant with the supplied data in the request body, and returns the new restaurant **(ALL FIELDS MUST BE PRESENT AND CORRECT TYPE).**

You should expect the following JSON to be submitted in the request.body:

```
{
    name: "The Saffron Lounge",
    location: "New York City, New York",
    phoneNumber: "123-456-7890",
    website: "http://www.saffronlounge.com",
    priceRange: "$$$$",
    cuisines: ["Cuban", "Italian" ],
    serviceOptions: {dineIn: true, takeOut: true, delivery: false}
}
```

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 restaurant can be created successfully, you will return the newly created restaurant (as shown below) with a 200 status code.

```
{
    _id: "507f1f77bcf86cd799439011",
    name: "The Saffron Lounge",
    location: "New York City, New York",
    phoneNumber: "123-456-7890",
    website: "http://www.saffronlounge.com",
    priceRange: "$$$$",
    cuisines: ["Cuban", "Italian" ],
    overallRating: 0
    serviceOptions: {dineIn: true, takeOut: true, delivery: false}
    reviews: []
}
```

`GET /restaurants/{id}`

Example: `GET /restaurants/507f1f77bcf86cd799439011`

Responds with the full content of the specified restaurant. So you will return all details of the restaurant. Your function should return the restaurant _id as a string, not an object ID

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

You will return the restaurant (as shown below) with a 200 status code along with the restaurant data if found.

```
{
    _id: "507f1f77bcf86cd799439011",
    name: "The Saffron Lounge",
    location: "New York City, New York",
    phoneNumber: "123-456-7890",
    website: "http://www.saffronlounge.com",
    priceRange: "$$$$",
    cuisines: ["Cuban", "Italian" ],
    overallRating: 3.5
    serviceOptions: {dineIn: true, takeOut: true, delivery: false}
    reviews: [{_id: 609f1f77bcf86cd799439023, title: "Amazing" ,.... } ,.... ]
}
```

`PUT /restaurants/{id}`

Example: `PUT /restaurants/507f1f77bcf86cd799439011`

This request will update a restaurant with information provided from the PUT body. Updates the specified restaurant **by replacing** the restaurant with the new restaurant content, and returns the updated restaurant. **(All fields need to be supplied in the request.body, even if you are not updating all fields)**

You should expect the following JSON to be submitted:

```
{
    name:  "Saffron Lounge",
    location: "SoHo, New York",
    phoneNumber: "123-456-1234",
    website: "http://www.thesaffronlounge.com",
    priceRange: "$$$",
    cuisines: ["Italian"],
    serviceOptions: {dineIn: false, takeOut: false, delivery:true }
}
```

reviews should not be able to be modified in this route. You must copy the old array of review ids from the existing restaurant first and then insert them into the updated document so they are retained and not overwritten. You should also not modify or overwrite the overallRating field, so copy that as well.

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 restaurant exist with an `_id` of `{id}`, return a 404 and end the request.

If the update was successful, then respond with that updated restaurant (as shown below) with a 200 status code

```
{
    _id: "507f1f77bcf86cd799439011",
    name: "Saffron Lounge",
    location: "SoHo, New York",
    phoneNumber: "123-456-1234",
    website: "http://www.thesaffronlounge.com",
    priceRange: "$$$",
    cuisines: ["Italian" ],
    overallRating: 3.5
    serviceOptions: {dineIn: false, takeOut: false, delivery: true}
reviews: [{_id: 609f1f77bcf86cd799439023, title: "Amazing" ,.... } ,.... ]
}
```

`DELETE /restaurants/{id}`

Example: `DELETE /restaurants/` `507f1f77bcf86cd799439011`

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

Deletes the restaurant, sends a status code 200 and returns:

```
{"restaurantId": "507f1f77bcf86cd799439011", "deleted": true}.
```

routes/reviews.js

You must do FULL error handling and input checking for ALL routes! checking if input is supplied, correct type, range etc. and responding with proper status codes when you encounter bad input. All the input types, values and ranges that apply to the DB functions apply to the routes as well so you will do all the same checks in the routes that you do in the DB functions before sending the data to the DB function

`GET /reviews/{restaurantId}`

Example: `GET /reviews/` `306f1f77bcf86cd799431423`

Getting this route will return an array of all reviews in the system for the specified restaurant id.

If no reviews for the `restaurantId` are found, you will issue a 404 status code and end the request.

if the `restaurantId` is not found in the system, you will issue a 404 status code and end the request

You will return the array of reviews (as shown below) with a 200 status code along with the review data if found.

```
[{
     _id: "603d992b919a503b9afb856e",
     title: "This place was great!",
     reviewer: "scaredycat",
     rating: 5,
     dateOfReview: "10/13/2021",
     review: "This place was great! the staff is top notch and the food was delicious!  They really know how
 to treat their customers"
}, .........
]
```

`POST /reviews/{restaurantId}`

Example: `POST /reviews/306f1f77bcf86cd799431423`

Creates a review sub-document with the supplied data in the request body, and returns the new review **(ALL FIELDS MUST BE PRESENT AND CORRECT TYPE).**

You should expect the following JSON to be submitted in the request.body:

```
{
  title: "This place was great!",
  reviewer: "scaredycat",
  rating: 5,
  dateOfReview: "10/13/2021",
  review: "This place was great! the staff is top notch and the food was delicious!  They really know how to
 treat their customers"
}
```

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

if the restaurantId is not valid (it cannot be found), you will issue a 400 status code and end the request.

If the JSON is valid and the review can be created successful, you will return all the restaurant data showing the reviews (as shown below) with a 200 status code.

```

{
    _id: "306f1f77bcf86cd799431423",
    name: "Black Bear",
    location: "Hoboken, New Jersey",
    phoneNumber: "456-789-0123",
    website: "http://www.blackbear.com",
    priceRange: "$$",
    cuisines: ["Cuban", "American" ],
    overallRating: 4
    serviceOptions: {dineIn: true, takeOut: true, delivery: true}
    reviews: [
    {
     _id: "603d992b919a503b9afb856e",

     title: "This place was great!",
     reviewer: "scaredycat",
     rating: 5,
     dateOfReview: "10/13/2021",
     review: "This place was great! the staff is top notch and the food was delicious!  They really know how
 to treat their customers"
},....]
}
```

`GET /reviews/review/{reviewId}`

Example: `GET /reviews/review/603d992b919a503b9afb856e`

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

You will return the review (as shown below) with a 200 status code.

```
{
     _id: "603d992b919a503b9afb856e",
     title: "This place was great!",
     reviewer: "scaredycat",
     rating: 5,
     dateOfReview: "10/13/2021",
     review: "This place was great! the staff is top notch and the food was delicious!  They really know how
 to treat their customers"
}
```

`Delete /reviews/{reviewId}`

Example: `DELETE /reviews/603d992b919a503b9afb856e`

Deletes the specified review for the specified `reviewId`, sends a 200 status code and returns:

`{"reviewId": "603d992b919a503b9afb856e", "deleted": true}`.

NOTE: If a review is deleted, you need to recalculate the average for the `overallRating` field in the main restaurant document

app.js

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

Tip for testing:

You should create a seed file that populates your DB with initial data for both restaurants and reviews. This will GREATLY improve your debugging as you should have enough sample data to do proper testing and it would be rather time consuming to enter a restaurants and reviews for that restaurant one by one through the API. A seed file is not required and is optional but is highly recommended. You should have a DB with at least 10 restaurants and multiple reviews for each restaurants for proper testing (again, this is not required, but it is to ensure you can test thoroughly.)

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 **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`