WEB422 Assignment 1

Submission Deadline:

Friday, January 21st @ 11:59pm

Assessment Weight:

5% of your final course Grade

Objective:

This first assignment will help students obtain the sample data loaded in MongoDB Atlas for the WEB422 course as well as to create (and publish) a simple Web API to work with the data.

Specification:

Step 1: Loading the "Sample Data" in MongoDB Atlas

The first step for this assignment is to create a new "Project" in your existing MongoDB Atlas account (if you have deleted your account from last semester, please revisit the documentation here from WEB322 - https://web322.ca/notes/week08).

Assuming that you have an account in MongoDB Atlas, please follow the instructions located below (from the WEB422 website) to create a new "Project", "Cluster" and "Load the Sample Dataset".

MongoDB Sample Data Instructions - https://web422.ca/notes/mongodb-sample-data

Step 2: Building a Web API

Once you have completed the guide (Step 1), and have the data loaded in a new Project within your MongoDB Atlas account, we must build and publish a Web API to enable code on the client-side to work with the data.

To get started:

- First create a folder (ie: "restaurantAPI") for your project somewhere on your machine. Next, open this folder in Visual Studio code and proceed to create a simple server using the Express framework. At this point only a single GET route "/" is required which returns the following object (JSON): {message: "API Listening"}. **NOTE**: This is to ensure that your environment is set up correctly and that you're able to run / test the server locally.

- Next, install the "cors" package using npm. This must be imported ("required") and used in its simplest form as a middleware function, declared before your routes, ie: **app.use(cors());**

- To ensure that our server can parse the JSON provided in the request body for some of our routes (declared below) we can use the express.json() built-in middleware (ie: **app.use(express.json())**).

- Finally, install the "mongoose" ODM using npm. This will be used by your "restaurantDB.js" module (to be downloaded shortly)

- Once you have installed your dependencies, initialize an empty Git repository for this folder using the command "git init"

Adding restaurantDB.js Module:

Now that your server is up and running, we must add the module that will provide the functionality to connect to the "restaurants" collection within the newly created "sample_restaurants" Database:

- First, create a "modules" folder within your solution to house the "restaurantDB.js" module.

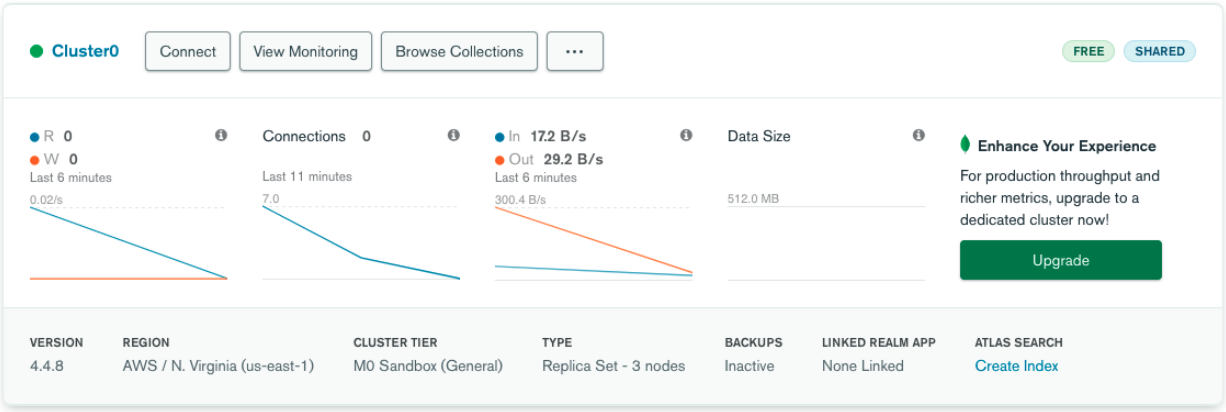
- Next, create the file "restaurantDB.js" within the "modules" folder and proceed to **copy the code from here.** (this is the completed "restaurantDB.js" module, for use with this assignment)

- Finally, back in your server.js file add the following lines to "require" the newly created "restaurantDB.js" module, as well as to create a new "db" instance to work with the data:

  ```
  const RestaurantDB = require("./modules/restaurantDB.js");
  const db = new RestaurantDB();
  ```

Obtaining the "MongoDB Connection String"

- On the MongoDB Atlas dashboard, ensure that you're looking at the overview for your newly created Cluster (within your newly created Project) that contains the sample data.



- Next, click the "CONNECT" button and grab the connection string using the "Connect Your Application" button. **NOTE:** If you have not yet created a user for this database, or whitelisted the ip: 0.0.0.0/0, please proceed to do this first.

- Once you have your connection string, it should look *something like this*:
  mongodb+srv://userName:<password>@cluster0-abc0d.mongodb.net/<dbname>?retryWrites=true&w=majority

- Next, replace the entire string <password> with your password for this cluster (ie: do not include the < & > characters)
- Finally, replace the text **<dbname>** with the database name: **sample_restaurants** and use the updated connection string to replace the text "Your MongoDB Connection String Goes Here" in your db.initialize() call (from below). *(See Step 4 below for info on a more secure way to handle this secure string in your server and environment)*

"Initializing" the Module before the server starts

To ensure that we can indeed connect to the MongoDB Atlas cluster with our new connection string, we must invoke the **db.initialize("connection string…")** method and only start the server once it has succeeded, otherwise we should show the error message in the console, ie:

```
db.initialize("Your MongoDB Connection String Goes Here").then(()=>{
   app.listen(HTTP_PORT, ()=>{
      console.log(`server listening on: ${HTTP_PORT}`);
   });
}).catch((err)=>{
   console.log(err);
});
```

Reviewing the restaurantDB.js Module (db)

This module will provide the 6 (promise-based) functions required by our Web API for this particular dataset, ie:

- **db.initialize(**"Your MongoDB Connection String Goes Here"**)**: Establish a connection with the MongoDB server and initialize the "Restaurant" model with the "restaurant" collection (used above)
- **db.addNewRestaurant**(**data**): Create a new restaurant in the collection using the object passed in the "data" parameter
- **db.getAllRestaurants**(**page**, **perPage, borough**): Return an array of all restaurants for a specific page (sorted by **restaurant_id**), given the number of items per page. For *example*, if **page** is **2** and **perPage** is **5**, then this function would return a sorted list of restaurants (by **restaurant_id**), containing items **6 – 10.** This will help us to deal with the large amount of data in this dataset and make paging easier to implement in the UI later.

  Additionally, there is an optional parameter "borough" that can be used to filter results by a specific "borough" value
- **db.getRestaurantById(Id)**: Return a single restaurant object whose "_id" value matches the "Id" parameter
- **updateRestaurantById(data**,**Id)**: Overwrite an existing restaurant whose "_id" value matches the "Id" parameter, using the object passed in the "data" parameter.
- **deleteRestaurantById(Id)**: Delete an existing restaurant whose "_id" value matches the "Id" parameter

Add the routes

The next piece that needs to be completed before we have a functioning Web API is to actually define the routes (listed Below). **Note**: Do not forget to return an error message if there was a problem and make use of the status codes 201, 204 and 500 where applicable.

- **POST /api/restaurants**

  This route uses the body of the request to add a new "Restaurant" document to the collection and return a success / fail message to the client.

- **GET /api/restaurants**

  This route must accept the numeric query parameters "page" and "perPage" as well as the string parameter "borough", ie: /api/restaurants?page=1&perPage=5&borough=Bronx. It will use these values to return all "Restaurant" objects for a specific "page" to the client as well as optionally filtering by "borough", if provided.

- **GET /api/restaurants**

  This route must accept a route parameter that represents the _id of the desired restaurant object, ie: /api/restaurants/ 5eb3d668b31de5d588f4292e. It will use this parameter to return a specific "Restaurant" object to the client.

- **PUT /api/restaurants**

  This route must accept a route parameter that represents the _id of the desired restaurant object, ie: /api/restaurants/5eb3d668b31de5d588f4292e as well as read the contents of the request body. It will use these values to update a specific "Restaurant" document in the collection and return a success / fail message to the client.

- **DELETE /api/restaurants**

  This route must accept a route parameter that represents the _id of the desired restaurant object, ie: /api/restaurants/5eb3d668b31de5d588f4292e. It will use this value to delete a specific "Restaurant" document from the collection and return a success / fail message to the client.

Step 3: Pushing to Heroku

Once you are satisfied with your application, deploy it to Heroku:

- Ensure that you have checked in your latest code using **git** (from within Visual Studio Code)
- Open the integrated terminal in Visual Studio Code
- Log in to your Heroku account using the command **heroku login**
- Create a new app on Heroku using the command **heroku create**
- Push your code to Heroku using the command **git push heroku master**

IMPORTANT NOTE: Since we are using an "**unverified" free** account on Heroku, we are limited to only **5 apps**, so if you have created 5 apps already, you must delete one (or verify your account with a credit card).

Assignment Submission:

1. Add the following declaration at the top of your server.js file

```
/*********************************************************************************
* WEB422 – Assignment 1
* I declare that this assignment is my own work in accordance with Seneca  Academic Policy.
* No part of this assignment has been copied manually or electronically from any other source
* (including web sites) or distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
* Heroku Link: _____
*
********************************************************************************/
```

2. Compress (.zip) the files in your Visual Studio working directory (this is the folder that you opened in Visual Studio to create your server-side code

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- Submitted assignments **must** run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.