WEB422 Assignment 1

Submission Deadline:

Friday, September 16th @ 11:59pm

Assessment Weight:

5% of your final course Grade

Objective:

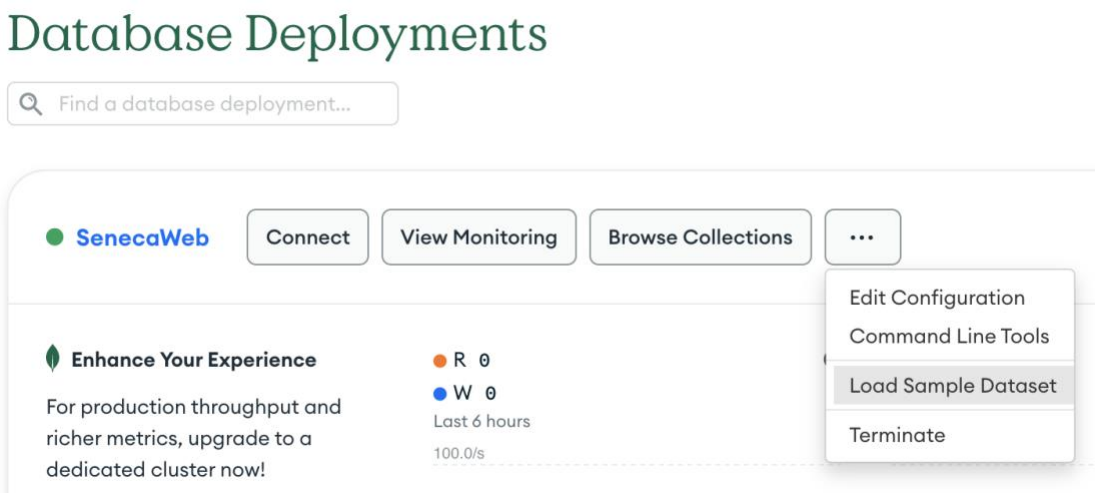
This first assignment will help students obtain the sample data loaded in MongoDB Atlas for the WEB422 course as well as to create (and publish) a simple Web API to work with the data.

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, proceed to load the “Sample Dataset” (Note: This operation takes approximately 5 minutes to complete):



(Please see the official documentation for more information)

Step 2: Building a Web API

Once you have Your sample dataset loaded into your cluster (Step 1), 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: "moviesAPI") 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());**

- Another package that we will need from npm is "dotenv" to enable our code to read from a ".env" file. This is also used in its simplest form, placed alongside your other require statements: **require('dotenv').config();**

- 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())**).

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

- Finally, create a **.gitignore** file at the root of your "moviesAPI" folder. The contents of this file should be:

  ```
  node_modules
  .env
  ```

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

Adding moviesDB.js Module:

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

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

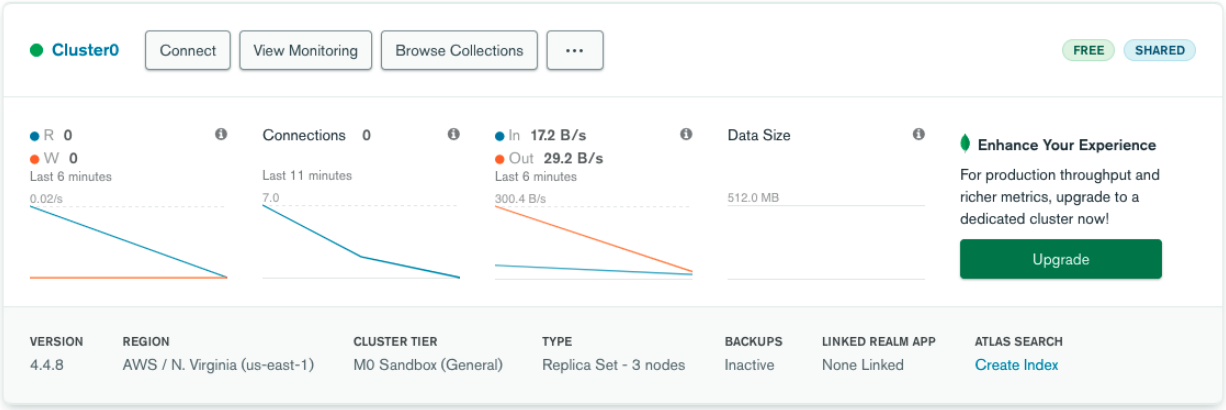
- Next, create the file "moviesDB.js" within the "modules" folder and proceed to **copy the code from here.** (this is the completed "moviesDB.js" module, for use with this assignment)

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

  ```
  const MoviesDB = require("./modules/moviesDB.js");
  const db = new MoviesDB();
  ```

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/?retryWrites=true&w=majority**
- Next, replace the entire string <password> with your password for this cluster (ie: do not include the < & > characters)
- Finally, add database name: **sample_mflix** *after* "mongodb.net/", ie: **… mongodb.net/sample_mflix?retryWrites …** and place the updated connection string in your **.env** file using the following syntax:

  MONGODB_CONN_STRING=**myUdatedConnectionString**

  Where **myUpdatedConnectionString** is your completed connection string (above)

"Initializing" the Module before the server starts

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

```
db.initialize(process.env.MONGODB_CONN_STRING).then(()=>{
   app.listen(HTTP_PORT, ()=>{
      console.log(`server listening on: ${HTTP_PORT}`);
   });
}).catch((err)=>{
   console.log(err);
});
```

Reviewing the moviesDB.js Module (db)

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

- **db.initialize**(**connectionString**): Establish a connection with the MongoDB server and initialize the "Movie" model with the "movies" collection (used above)
- **db.addNewMovie**(**data**): Create a new movie in the collection using the object passed in the "data" parameter
- **db.getAllMovies**(**page**, **perPage, title**): Return an array of all movies for a specific page (sorted by **year**), 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 movies (by **year**), 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 "title" that can be used to filter results by a specific (case sensitive) "title" value
- **db.getMovieById(Id)**: Return a single movie object whose "_id" value matches the "Id" parameter
- **updateMovieById(data**,**Id)**: Overwrite an existing movie whose "_id" value matches the "Id" parameter, using the object passed in the "data" parameter.
- **deleteMovieById(Id)**: Delete an existing movie 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/movies**

  This route uses the body of the request to add a new "Movie" document to the collection and return the newly created movie object / fail message to the client.
- **GET /api/movies**

  This route must accept the numeric query parameters "page" and "perPage" as well as the (optional) string parameter "title", ie: /api/movies?page=1&perPage=5&title=The Avengers. It will use these values to return all "Movie" objects for a specific "page" to the client as well as optionally filtering by "title", if provided (in this case, it will show both “The Avengers” films).
- **GET /api/movies**

  This route must accept a route parameter that represents the _id of the desired movie object, ie: /api/movies/573a1391f29313caabcd956e. It will use this parameter to return a specific "Movie" object to the client.

- **PUT /api/movie**

  This route must accept a route parameter that represents the _id of the desired movie object, ie: /api/movies/573a1391f29313caabcd956e as well as read the contents of the request body. It will use these values to update a specific "Movie" document in the collection and return a success / fail message to the client.

- **DELETE /api/movies**

  This route must accept a route parameter that represents the _id of the desired movie object, ie: /api/movies/573a1391f29313caabcd956e. It will use this value to delete a specific "Movie" document from the collection and return a success / fail message to the client.

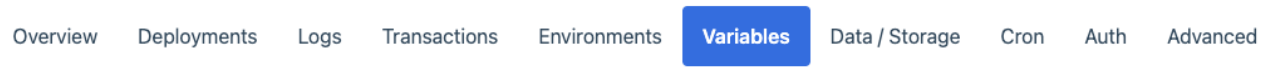
Step 3: Pushing to Cyclic

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

- For explicit instructions on publishing to Github / Cyclic, refer to the "Getting Started with Cyclic" guide on the WEB322 Site: https://web322.ca/getting-started-with-cyclic, beginning with "Create a GitHub Repository"

- After cyclic completes its build step, you will notice that it will prompt you to enter a value for MONGODB_CONN_STRING. This is where you will enter a value to match what is stored locally in .env

  NOTE: If you wish to come back later to edit your environment variables, they can be accessed via the "Variables" tab for your Cyclic deployment:



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: _____
* Cyclic 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).

   NOTE: Do **not forget** to include your **.env** file in your assignment submission. Without this file, the assignment will not run locally.

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.