COMP 2406 B - Fall 2022
Tutorial #8

Due Sunday, November 27, 23:59

Objectives

- Create an Express app that uses MongoDB for data storage
- Practice querying MongoDB from within an Express app

Expectations

You need to submit solutions to all the problems in this tutorial. Our TAs will mark your submission. Remember to use the available resources (w3schools, Node.js and MongoDB documentation, lecture materials, etc.) for more information if you are struggling to complete the problems.

Marking scheme. For each tutorial, you will receive:

- 2/2 for submitting high-quality solutions to all problems. "High-quality" means that your code works (solves the problem) and is also neat and concise (no overengineering, please).
- 1/2 for submitting solutions for all problems but some need improvement, or you are missing a problem.
- 0/2 if you are missing several problems or your solutions are poorly done; or you do not make a submission, or your submission cannot be executed.

Problem Background

The goal of this tutorial is to provide a card search service that allows the user to specify a filter and have the server return cards matching that filter. This tutorial continues to work with the dataset from tutorial #5. You do not need to have completed tutorial 5 in order to start this tutorial. A summary of the fields available within each object of the dataset is provided at the end of this document. Download the **T08-BaseCode.zip** file from Brightspace to get started. This zip file contains the following files:

1. database-initializer.js: Run this file while your Mongo daemon is running and it will create a database called 't8' with a collection called 'cards' that contains all of the cards from the dataset. **It would be good to review this file to get an idea of how to create your own database initializer script for your assignment 4.**
2. server.js: This file contains an Express app that provides some base functionality for you. When you run this file, it will connect to the Mongo database and initialize a variable called

db representing the '`t8`' database. You can use this variable to query the database throughout your app. The server will also serve a default index page using Pug in response to **GET** requests for the route **/**, and respond to **GET** requests for the parameterized URL **/cards/*{cardID}*** when given a valid ID from the database.

3. **views/index.pug**: The template file used to render the default index page. You can add any additional elements that are necessary to this file as you progress through the tutorial.
4. **views/card.pug**: The template file used for rendering single card pages.

Problem 1 (Finishing the Index Page)

Run the **server.js** file and load the index page. The class and rarity drop-down menus do not currently have any values contained within them. Add code so that each of the drop-down menus has a complete list of all classes/rarities within the dataset. To find all of the classes/rarities, you can use the `db.collection.distinct("fieldName")` method within MongoDB. For example, `db.cards.distinct("cardClass")` will give you an array of the unique card classes within the database (note: this is the syntax for Mongo command line and the Node.js syntax will be different). **Remember that these methods are asynchronous in Node.js – you need to specify callback functions!** You may use Pug to generate the proper HTML code to fill in the dropdown values or have your server support requests for the class/rarity information and use AJAX requests when the page loads to retrieve the dropdown values.

Problem 2 (Querying for Class/Rarity)

The next step will be to allow the client to use the index page to query cards based on class and/or rarity. One way to do this is to add client-side JavaScript capable of querying the server and performing page updates. An alternative approach would be to modify the index page to have a form that would allow the client to submit a request to the server using the “Refresh Results” button and load an entirely new page containing the results. You are free to take either approach, but only the client-side JavaScript approach will be explained in more detail here.

Start by creating a client-side JavaScript file and adding a reference to that file into the index page template file. Update your server so that the client JavaScript can be loaded successfully. The client-side JavaScript should add a click handler to the “Refresh Results” button. When the “Refresh Results” button is clicked, you can create an XMLHttpRequest including the selected class/rarity values from the page and send it to the server. The server can respond with a list of matching cards.

To facilitate this, add a route on your server to handle **GET** requests to the URL **/cards**. Add code to handle query parameters so that card class and rarity values can be specified in the request. Add code into the route handler to build a query document using the query parameters (e.g., card class and rarity), query the database for matching cards, and send a response to the client containing the matching cards.

There are multiple ways you can send the response back to the client. First, you could send JSON data to the client and have the client-side JavaScript update the HTML on the page (like earlier tutorials). An alternative approach is to render a partial page using Pug and send that HTML to the client directly. If you choose the Pug approach, create a new Pug template file that will accept an array of card objects as input. This template should produce a partial page (e.g., no html/head/body elements) that contains a list of the matching cards. Each card entry should be a link to that specific card page (use the _id field from the database) and the text of the link should be the card name. When the client-side JavaScript receives this HTML in response to its XMLHttpRequest, it can update the HTML of the results div on the page.

Test out your code before proceeding. For reference, the correct card name results for class="MAGE" and rarity="LEGENDARY" are: Rhonin, Stargazer Luna, Inkmaster Solia, Archmage Antonidas, Toki, Time-Tinker, Archmage Arugal, Flame Leviathan, Sindragosa, Dragoncaller Alanna, Anomalus, Pyros, and Pyros. You can also click each card you get in your result list to ensure the values of the card match the search query.

Problem 3 (Adding Other Query Parameters)

Now add support for the remainder of the query parameters:

1. Min Attack/Health: These are integer values. A card should match if its attack/health property is greater than or equal to the query parameter value.
2. Max Attack/Health: These are integer values. A card should match if its attack/health property is less than or equal to the query parameter value.
3. Name/Artist: These are string values. A card should match if the card name/artist value contains the given value. Additionally, you should ensure that the search is done in a case-insensitive matter. You can use a `$regex` query operator in MongoDB to perform a string search of these fields.

It will be easiest to add support for the integer parameters first. Note that since a card must match all specified query parameters, you can use a single $and query with an array containing an entry for each query parameter that has been specified in the request.

Summary of Fields in Card Documents:

Each card in the database has the following field names:

1. _id: the unique ID created by MongoDB when the card was inserted
2. artist: a string representing the name of the artist(s) that created the card art
3. name: a string representing the name of the card
4. cardClass: a string representing the class of player that can use the card
5. rarity: a string representing the rarity of the card
6. attack: an integer representing the attack value of the card
7. health: an integer representing the health value of the card

Additional fields have been removed from the dataset, as they are not necessary for this tutorial.

Problem 4 (Save Your Work & Submit)

Once you have completed all the problems for this tutorial, **zip** (compress) all your files into a single **.zip** file and submit it to the Tutorial submission on Brightspace. Name your file **T8-YourName.zip**. Your dependencies must be installable via the **npm install** command. You should not include your **node_modules** folder in your submission. You must include a **README** file with detailed instructions on how to run your server and test your application. Make sure you download your .zip file and check its contents after submitting. If your .zip file is missing files or corrupt, you will lose marks.