Final Report: “Show Track” App

By “NoSQL Nomads”
(Mohan Chimata, John Jeffery)

[Source code: https://github.com/mohanch16/ShowTrack]

Overview:

This project consists of a web application called Show Track. Show Track allows users to view the catalog of television shows and movies on Disney+, Amazon Prime Video, and Netflix, allowing users to view, search, and filter through these catalogs based on attributes such as title, show-type, and video subscription-provider that the show is available in, etc.

These datasets originate from Kaggle as three separate collections, yet were altered and integrated into one dataset for this program. Show Track is a full stack web app utilizing HTML, CSS, Blazor WebAssembly (C#) for the front end page design and graphical user interface, while the back end utilizes C# with .Net Core and MongoDB as the database. The database contains an organized and updated dataset and connects to the project using the MongoDB driver for .Net.

A “watch party” styled chat feature has also been prototyped for reviews and show discussion, preliminary highly-secure proxy shows-discussion feature has been released where users can discuss and decide on shows, using the .Net SignalR Client for Blazor.

Setup and Installation:

1. Download and install the relevant version of dotnet core SDK (software development kit) based on your operating system as instructed in the microsoft dotnet core download page. (https://dotnet.microsoft.com/en-us/download)
2. Make sure the dotnet core CLI (command line interface) has been installed by typing "dotnet --version" in the terminal.
3. IDE Visual studio code and C# extensions: For code walkthrough and editing IDE such as Visual studio code (download for your operating system from this link) is recommended, you can install extensions for C# and blazor from left-nav bar extensions option. This IDE also lets you use an intellisense feature which shows errors, make suggestions based on the code while developing.

Building and running the application:

1. To run and build the application, open to visual studio code terminal (VS code top menu bar >> view >> terminal, or shortcut: command[for mac]/windows key + `).
2. Use command: dotnet watch —project Server

Implementation and Functionality:

(a) Database design:

The Show Track web application uses a MongoDB database to store the catalogs of movies and tv shows across three media platforms. Initially, the data consisted of three separate collections, one for each service. However, to avoid joins and optimize query performance, these data sets have been

merged into one collection, with an additional field added to specify which dataset each document originates from. In addition, several fields and attributes have been adjusted. Every field title was renamed for better user experience as well as better integration with the backend Blazor (That is, for example, listed_in to Categories). The fields Cast, Directors, and Categories have also been updated from a single string entry to an array of substrings. This was done using a Mongo shell command to scan through each of these fields and divide the string value into substrings, saving each substring to an entry within the field's array. The values for subscription service and media type (movie or tv show) were also changed to be enumeration values. This allows faster query time as instead of needing to compare each character within a string for a matching entry, it can instead compare a single integer for each document. A list of some of the main Mongo Shell commands can be seen below.

```
SPLITTING TO ARRAY
 Catalogs> db.FullCatalog.aggregate([ { $project: { categories: { $split: [ "$categories", ", "
]}}}, {$merge: "FullCatalog"}])

CHANGE FIELD VALUE
db.FullCatalog.updateMany({"SubscriptionType": "Disney"}, {"$set": { "SubscriptionType": "3"}})

RENAME FIELD
db.FullCatalog.updateMany({}, {$rename: {'country': 'Country'}}, false, true)

REMOVE FIELD
 db.FullCatalog.updateMany({}, { $unset: {Id: "" }} )

COMBINE ARRAY TO STRING
db.FullCatalog.aggregate([ { $match: { categories: { $type: "array" } } }, { $addFields: {
categories: { $arrayElemAt: ["$categories", 0] } } }]). forEach((doc) => db.FullCatalog.updateOne({
_id: doc._id }, { $set: { categories: doc.categories } }));
```

Figure: *Four of the main commands crucial to updating the dataset.*

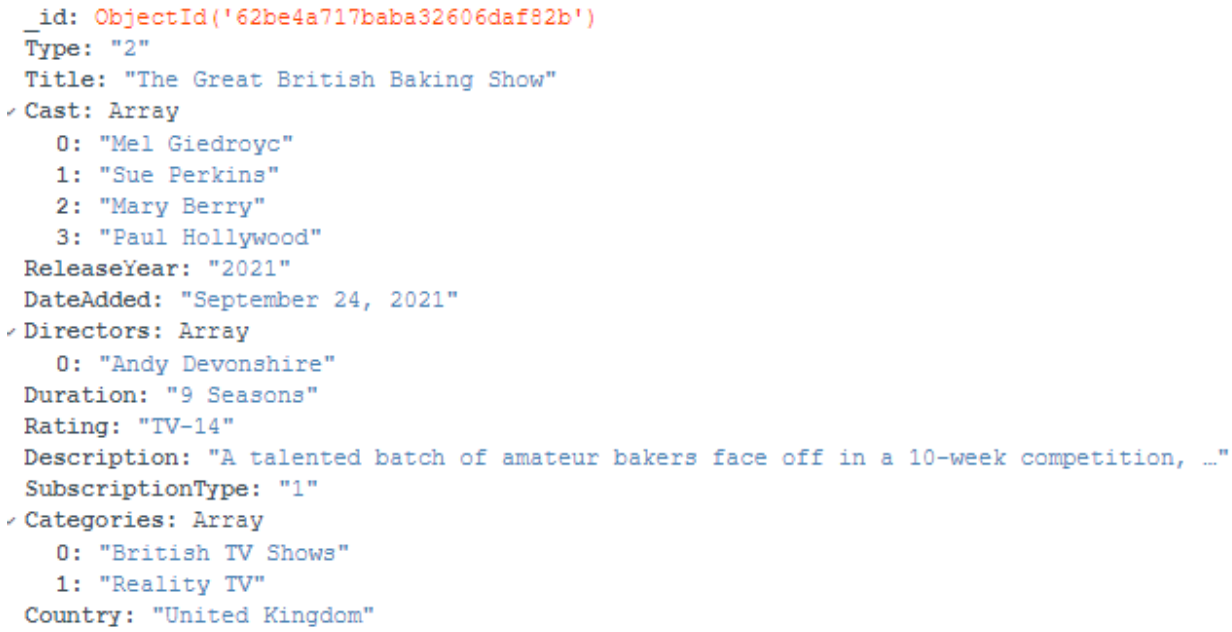

Figure: *The current iteration of a document entry, showing use of enums and arrays.*

(b) Backend Implementation and Database connections:

(i) Data models mapping to and from database:

Data is sent between the backend server and the database via mapping of C# models. When data documents are sent using the model, they require being serialized in either json or bson formatting. These models become embedded mongodb bson documents when used with write operations, and are reverted back to C# classes when a read operation occurs.

This serialization and deserialization occurs using in-built attributes within the mongodb driver such as called bson id, bson-representation and

bson-type object-id. This bson representation with parameter as object id, lets the mongodb driver use the object-id type for id attribute, bson id attribute enables the mongodb to treat the id field on which attribute is declared as primary key and generates a unique id. Hence, mongodb driver automatically serializes the id field to and from the object-id type between mongodb and string type in dotnet core. Similarly, all other fields including the arrays serializes to or deserializes back from bson arrays to c# string array types.

(ii) Backend business logic:

All database operations are performed using the MongoDB driver for dotnet which lets users use the MongoDB query API functions which are almost similar across the platforms. In addition to the regular API, for this CSharp driver, it provides a fluent API which follows the builder pattern and allows the user to follow either option. This application uses the common query API for CRUD (create, read, update and delete) operations, and utilizes the builder pattern based LINQ queries for Search, filter and pagination options.

(iii) Architecture:

The backend uses the Blazor .Net template in a standard MVC (model-view-controller) architecture, where ShowTrack.Client is the view component, to facilitate communication between the database and the front end of the web application. The Client subdirectory contains razor files with

C# and html code for each of the web pages used in the application. The client section also contains the imports necessary to use the other two directories of the application. While initially the Blazor server hosting model was used, the project was migrated to Blazor webassembly due to better organization as well as code sharing potential. The ShowController file contains the class for the ShowsController as well as the functions to be used by the application, such as the get, post, update, delete, and search operations. The Model meanwhile contains the declaration for the Show class, with fields pertaining to each field title in the documents. Vital to the project is the Program.cs file, which functions as the core of the entire application. This calls the Main function and configures the various services needed to build and launch the program.

(d) Filters and Server-side pagination: Since the dataset consists of around 20,000 records of shows data, dumping the entire dataset into the (user interface) UI makes the page loads slow and daunting for the end-user to navigate through the application. Keeping this problem in mind, Server-side pagination and filtering has been implemented. That means, Based on the filters including search string, the data will be loaded by a particular page size which is ten by default. Hence, users will only see the first 10 shows in the UI but can change the page size, or navigate through all available pages using the pagination section located in the bottom of the application screens.

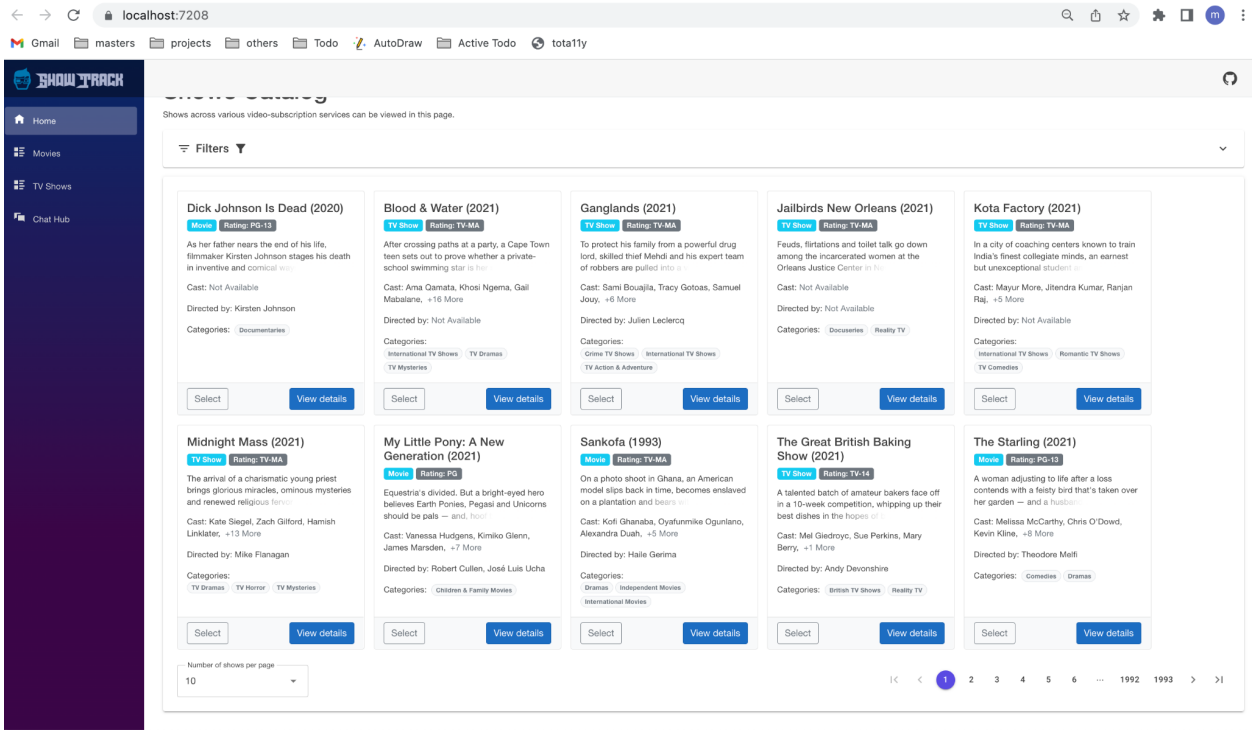

Figure: *Home page screen showing the pagination section from where users can change page-size, or navigate through pages and view the content of selected page-size loaded by server-side pagination.*

(c) Swagger for API Endpoints testing and admin Ops:

Swagger UI is a separate application that can be integrated with the built web api to test the various operations that a user would perform. These functions are get, post, delete, and put as seen below.

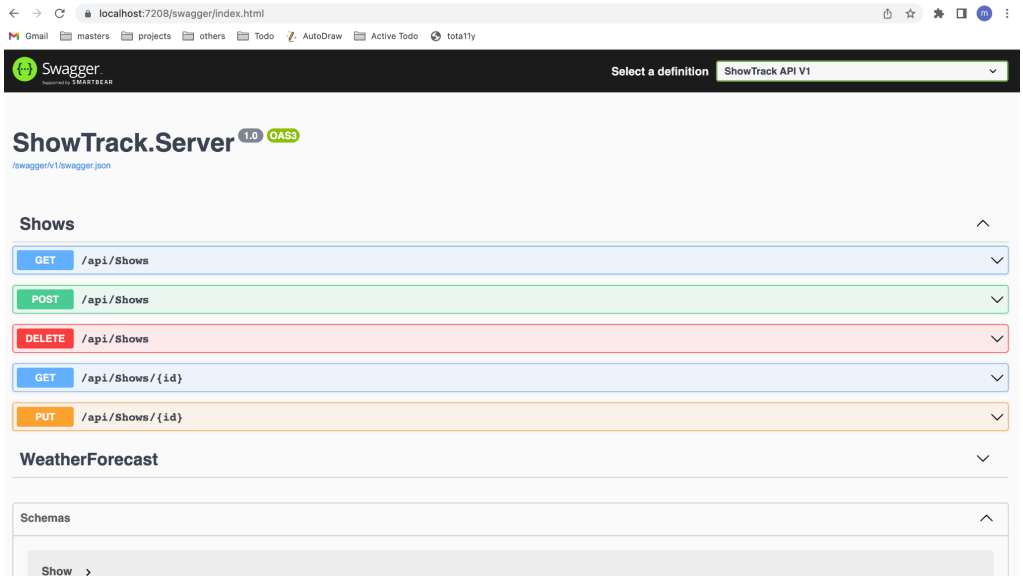

Figure: *Check the swagger integration page from where the built web api can be tested / can be used to mimic the front-end requests. Notice the "swagger/index.html" route part in the url after the website domain.*

(d) Frontend Implementation and features:

The frontend of the Show Track web application utilizes C# and Blazor to integrate with the back end as previously stated. The home page displays a small portion of the database entries and their details. Separate pages exist for the tv shows and movies, which help to filter results to the respective media type. The search feature functions to query based off of the title input into the search field. The UI for the main home page can be seen in the image below.

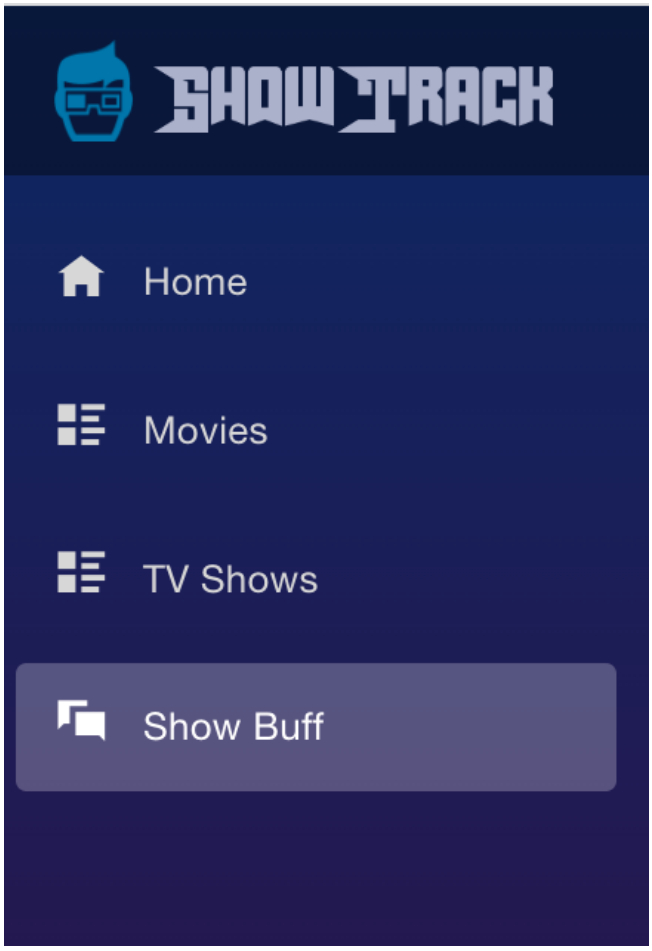

Figure: *Our ShowTrack logo and brand with latest menu*

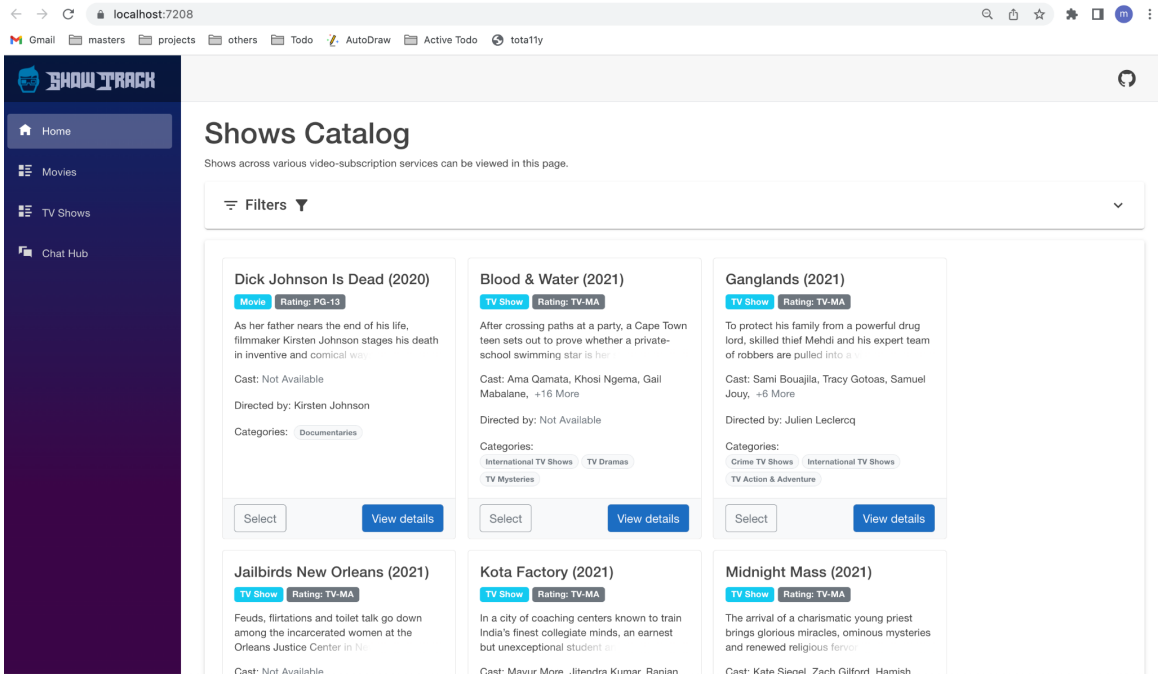

Figure: Application home page with "Our logo and brand" and filters option minimized.

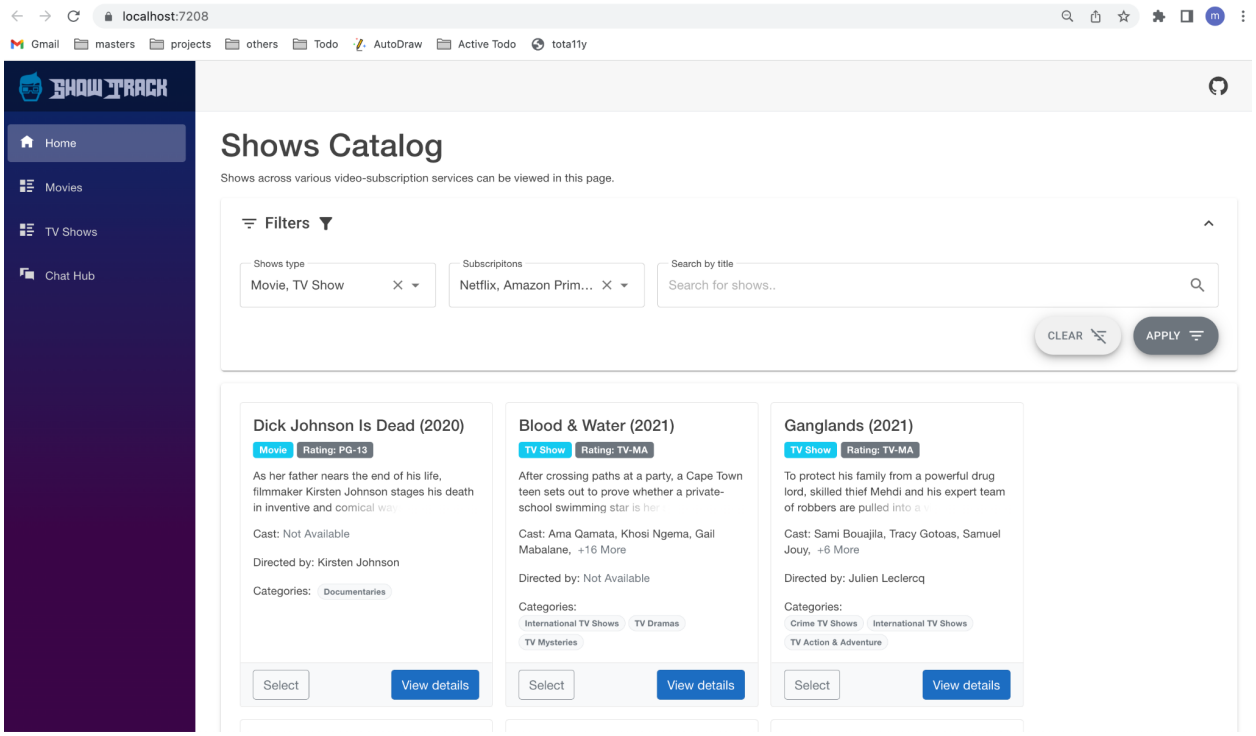

Figure: The main home page of the Show Track application. Note the filters for type and subscriptions and the search by title box.

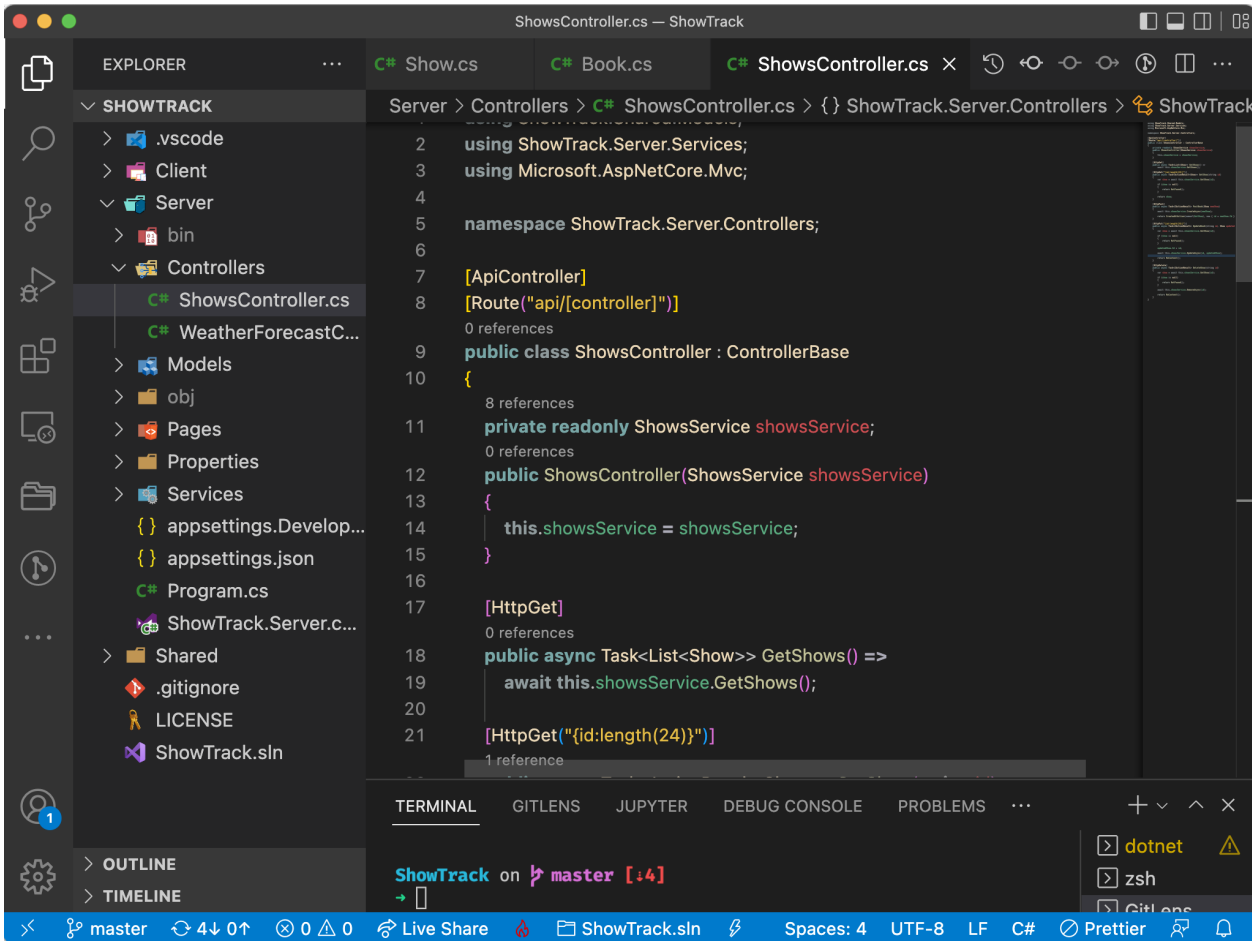

Figure: *Project structure (with Blazor-webassembly hosted-model) with the webapi controller code.*

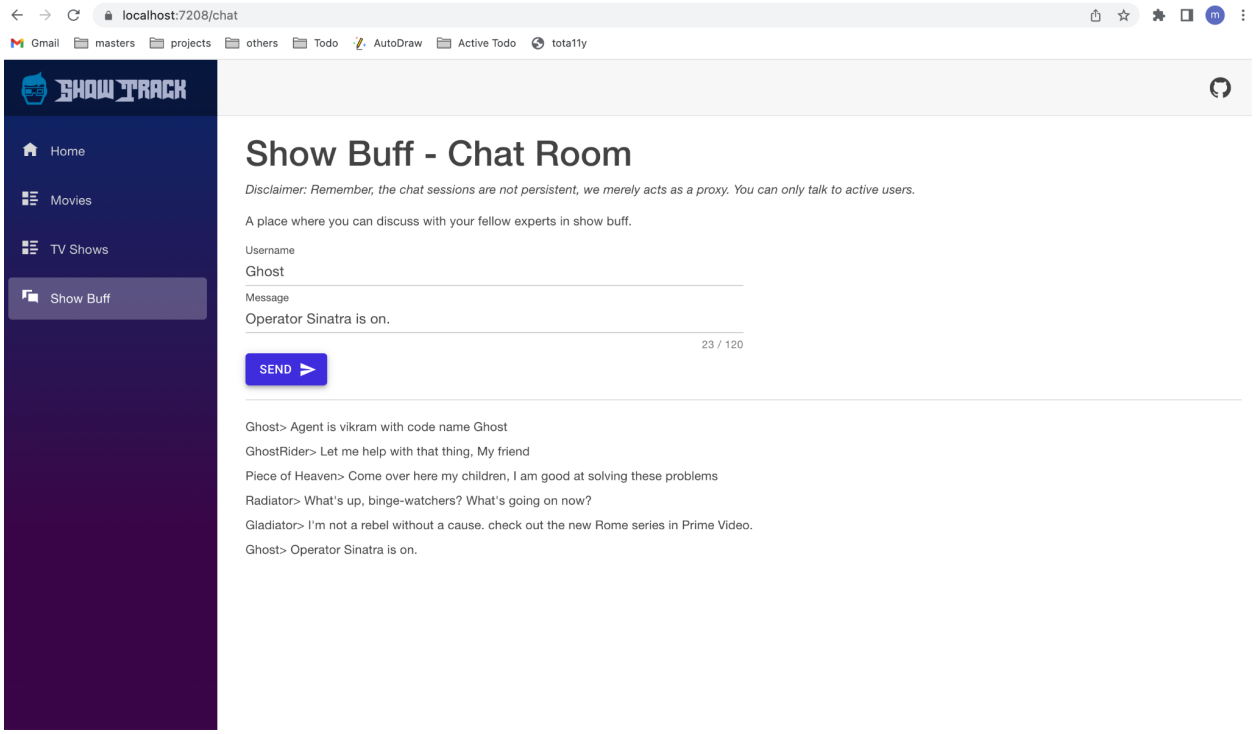

Figure: *"Show Buff" tab where users can chat anonymously with a name and message with fellow active users. Our application is merely a proxy, do not store the chat data. This page is designed to make a conversation with other users to discuss shows.*

Future Work:

The future tasks required for this project are as follows:

- Improving efficiency of search and filter query operations by adding a few more database indexes such as partial/sparse indexes for better query performance which also helps in handling more data efficiently in the future.
- Building the user interface with input control forms for admin operations, and integrating it with the WebApi.
- Integrate the watch party chat feature into the main program.
- Implementing select and compare shows feature, and few more filter options such as genres, and release-year range slider.

- Re-evaluating the performance of major functionalities such as search, and filter operations, and using best practices to improve the performance if necessary.
- Testing the functionalities, and fixing bugs in an iterative approach.

Challenges:

Following are prominent challenges experienced during development:

- Finalizing project architecture: Blazor hosting model (defines how blazor works and is structured) vs dotnet core web-api with react/angular, had to work on a significant number of sample applications to understand whether one works for this use case given the time constraints.
- Model mapping: This took a lot of time especially due to the complex types such as collections, nested-documents in the model and unclear and relatively less number of examples of mongodb csharp driver model mapping. Had to do the model redesigning to make the mapping job easy and dodge the existing issues, though it might seem simple.
- Filters and interactive features implementation issues: Since we are familiar with bootstrap for styling which was already scaffolded in when we created the blazor, we thought, for filters, we can utilize existing dropdowns in the bootstrap by few customizations, or by adding form-controls through a modal popup window.
    - Unfortunately, we didn't realize that such features involve the javascript dependencies of bootstrap which was not included in the scaffolded project, as blazor might work as unexpected when working along with javascript in some cases.
    - So, we had to find the quickest and best way to develop the UI. We found a bunch of libraries/Nuget (package manager for

dotnet) packages, and finalized “MudBlazor” as it is based on Material UI (UI components by google similar to bootstrap) for Blazor, and has complete components.

- Problem with selecting a new components library just a day before project submission is the learning curve, usage, impacts on existing user interface. For filters, we faced a lot of issues such as the filters component (child) is not triggering the event updates and passing the data to the main content component.

Contributions:

1. Database setup, datasets importing, integrating the datasets, data cleaning, database operations, building database indexes and performance moderating - **John Jeffery**.

2. Backend implementation, and Front end UI Implementation, Integrating backend to database, Swagger UI - **Mohan Chimata**

3. Database design, UI design, query design and filtering finalizing, testing application features - **John Jeffery, Mohan Chimata**.

References:

1. Netflix data dataset (https://www.kaggle.com/datasets/shivamb/netflix-shows)

2. Disney+ dataset (https://www.kaggle.com/datasets/shivamb/disney-movies-and-tv-shows)

3. Amazon prime dataset (https://www.kaggle.com/datasets/shivamb/amazon-prime-movies-and-tv-shows)

4. MongoDB (https://www.mongodb.com/)

5. Blazor - free and open source full-stack framework for developing single-page-applications through C#. (https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor)

6. MudBlazor - UI components library for Blazor framework (https://mudblazor.com/)

7. SignalR client library for Blazor (Chat hub creation tutorial in blazor through SignalR by Microsoft).

8. MongoDB CSharp / .NET Driver (https://www.mongodb.com/docs/drivers/csharp/)