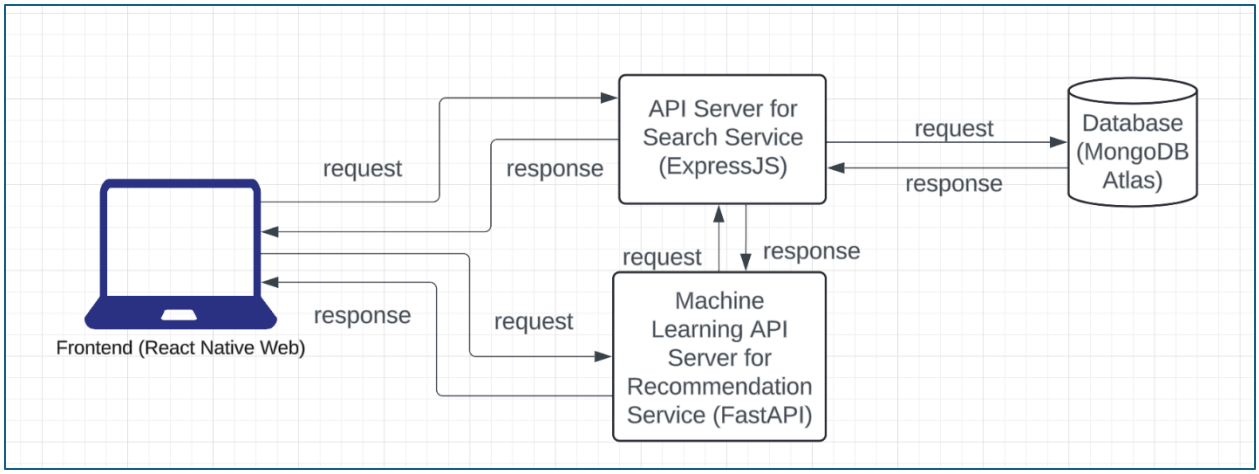
DIT 637 Smart and Secure Systems
TT06A Experiencing ML
07/31/2024 Developed by Clark Ngo
07/31/2024 Reviewed by Sam Chung
School of Technology & Computing (STC) @ City University of Seattle (CityU)

We use the same application that we started in TT 04.

References

- FastAPI https://fastapi.tiangolo.com/
- scikit-learn https://scikit-learn.org/stable/
- Cosine Similarity https://www.datastax.com/guides/what-is-cosine-similarity

Key Concepts and Tools for Experiencing ML



Frontend

To create the visual part of a mobile app that users interact with

- **React Native**: To efficiently build dynamic and responsive user interfaces for mobile apps
  - **React Native Web: an open-source library that allows React Native components and APIs to run on the web. It provides a way to use React Native's components and styling on web applications, enabling developers to write a single codebase that can run on both mobile and web platforms.**

Backend

The backend of a web application is responsible for the server-side logic, handling database interactions, user authentication, and application logic. It processes incoming requests, applies business rules, accesses databases, and sends responses back to the client.

Search Service API Server

- **ExpressJS**: A minimal and flexible Node.js web application framework that provides robust features for web and mobile applications. It is used to build the server-side logic of web applications.
  - **Pagination: Pagination is a technique used to divide a large set of data into smaller, manageable chunks, called "pages." This method is commonly used in web applications**

and user interfaces to improve usability and performance by displaying a subset of data at a time rather than loading all data at once.

Recommendation Service Machine Learning API Server

- **FastAPI:** A modern, fast (high-performance) web framework for building APIs with Python 3.6+ based on standard Python type hints. It's designed for ease of use, offering automatic interactive API documentation, high speed, and asynchronous support. FastAPI simplifies building robust APIs with minimal code and clear syntax.
    - **Sckit-learn:** is a popular machine learning library in Python, and one of the features it provides is the computation of similarity metrics, including cosine similarity.
    - **Cosine similarity:** is a measure of similarity between two non-zero vectors of an inner product space. It is defined as the cosine of the angle between two vectors, which can be interpreted as the normalized dot product of the vectors.

Database

A database is an organized collection of data generally stored and accessed electronically from a computer system. It is essential for storing and managing large amounts of data efficiently. There are two types: SQL and NoSQL DB.

- **MongoDB Atlas**: A cloud-based NoSQL database service for modern applications, providing a fully managed and global cloud database with built-in automation for resource and workload management.

Microservices

Microservices is an architectural style for developing software systems, where an application is composed of small, independent services that communicate with each other over a network, typically using HTTP or messaging protocols. Each microservice focuses on a specific business function and is designed to be loosely coupled, highly cohesive, and easily deployable.

Development Environment

Codespaces: To provide a cloud-based development environment using containerization that is easy to set up and use from anywhere, similar to Google Docs, but for coding.

Version Control and Collaboration

GitHub as Repo: To store and manage code with version control, making collaboration more manageable, just like using Google Drive for sharing documents.

Example: Movie Recommendation and Search Application

To practice and demonstrate the capabilities of React Native, ExpressJS, FastAPI, and sckit-learn in creating real-world applications, like recommending and searching for movies on Netflix.

Why do we have another backend server FastAPI while we have ExpressJS?

Using FastAPI for recommendation logic and ExpressJS for database connections allows for **a separation of concerns** and leverages the strengths of both frameworks. FastAPI is efficient for handling complex computations, such as recommendation algorithms, due to its asynchronous capabilities and modern Python syntax. ExpressJS, being a Node.js framework, can efficiently manage database interactions, providing a robust and scalable solution. This separation also enables modular development, easier maintenance, and independent scaling of services based on their specific requirements.

1.

Why Do We Need Machine Learning for Movie Recommendation?

Machine learning is used in movie recommendation systems for several reasons:

1. **Personalization**: Machine learning algorithms can analyze user preferences, past behaviors, and interactions to provide personalized recommendations. This means users get suggestions tailored to their tastes, which improves their experience and engagement.
2. **Handling Large Datasets**: Modern recommendation systems deal with vast amounts of data, including user ratings, movie metadata, and user profiles. Machine learning can process and analyze this data efficiently to make accurate recommendations.
3. **Discovering Patterns**: Machine learning algorithms can uncover complex patterns and relationships in data that traditional methods might miss. For example, they can identify similarities between movies that are not immediately obvious, like thematic or stylistic similarities.
4. **Scalability**: As the number of users and movies grows, machine learning systems can scale to handle increased complexity and volume of data without requiring manual adjustments.
5. **Improving Over Time**: Machine learning models can continuously learn and adapt based on new data. As users interact with the system, the recommendations become more accurate and relevant over time.

2. Why Do We Use Cosine Similarity and Count Vectorizer?

Cosine Similarity and **Count Vectorizer** are used in text-based movie recommendation systems to measure similarity between text documents (such as movie descriptions or user reviews).

Cosine Similarity

- **Purpose**: Measures the similarity between two non-zero vectors. In the context of text, it calculates how similar two documents are based on their term frequencies.
- **Why It's Used**:
    - **Magnitude Independence**: Cosine similarity is useful because it normalizes the length of the vectors (documents), so the similarity measure is independent of the document length. This is important because longer documents might have higher term counts but not necessarily be more relevant.
    - **Angle Measurement**: It measures the cosine of the angle between two vectors. If the angle is small, the cosine similarity is high, indicating that the documents are similar. This helps in identifying documents (movies) that are conceptually similar.

Count Vectorizer

- **Purpose**: Converts a collection of text documents into a matrix of token counts. Each document is represented as a vector where each dimension corresponds to a term from the entire vocabulary, and the value represents the count of that term in the document.
- **Why It's Used**:
    - **Feature Extraction**: Count Vectorizer transforms text data into numerical features that can be used by machine learning algorithms. It creates a matrix where each row represents a document and each column represents a term.
    - **Simple and Effective**: It's a straightforward method for text representation. Despite its simplicity, it's effective for many text-based tasks, including document similarity and recommendation.
    - **Baseline Model**: It provides a baseline model for more complex text representations like TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings.

Uploading three image files to your GitHub Repository generated from GitHub Classroom

1. The screenshot of your 'Initial Recommendations' as 'firstname_lastname_initial_recommendations.png'.
2. The screenshot of your 'API URLS' as 'firstname_lastname_api_urls.png'.
3. The screenshot of your 'Movie Recommendation' as 'firstname_lastname_movie_recommendation.png'.

FAQs

What environment variables (.env) should I have for backend-fastapi? (Sample)
Note: make sure there's no "/" at the end of the url.

Correct:

Wrong:
EXPRESSJS_BASE_URL=http://34.25.57.88:3000/

1) User Case

As a movie enthusiast using a **mobile device**, I want to search and browse a list of movies with details such as title, genre, and year, so that I can easily find information about movies I am interested in **while on the go**.

As a movie enthusiast using a **mobile device**, I want to get movie recommendations from my selections.

2) Code Overview

Search Service API Server with ExpressJS

Imports and Setup

```
import express from 'express';
import mongoose from 'mongoose';
import dotenv from 'dotenv';
import cors from 'cors';
import bodyParser from 'body-parser';
import morgan from 'morgan';
```

- **express**: A web framework for Node.js that simplifies routing and server setup.
- **mongoose**: An ODM (Object Data Modeling) library for MongoDB and Node.js, used to interact with MongoDB.
- **dotenv**: Loads environment variables from a .env file into process.env.
- **cors**: Middleware to enable Cross-Origin Resource Sharing (CORS), allowing requests from different origins.
- **bodyParser**: Middleware to parse incoming request bodies, in this case, JSON data.
- **morgan**: HTTP request logger middleware for Node.js.

Configuration

```
dotenv.config();

const app = express();
const port = 3000;
```

- **dotenv.config()**: Loads environment variables from the .env file.
- **app**: Creates an Express application.
- **port**: Specifies the port on which the server will listen.

Middleware Setup

```
app.use(cors()); // Enable CORS
app.use(bodyParser.json()); // Middleware to parse JSON request bodies
app.use(morgan('dev')); // Log HTTP requests
```

- **cors()**: Adds CORS headers to responses, allowing cross-origin requests.
- **bodyParser.json()**: Parses JSON request bodies into JavaScript objects.
- **morgan('dev')**: Logs HTTP requests in a "dev" format, useful for debugging.

MongoDB Connection Setup

```
const uri = process.env.MONGODB_URI;
if (!uri) {
    console.error('MongoDB URI is not defined');
    process.exit(1);
}

let isConnected = false;

const connectToMongoDB = async (req, res, next) => {
  if (!isConnected) {
    try {
      await mongoose.connect(uri, {
        dbName: 'sample_mflix' // Set the database name
      });
      isConnected = true;
      console.log('Connected to MongoDB Atlas');
    } catch (error) {
      console.error('Error connecting to MongoDB Atlas:', error);
      return res.status(500).json({ error: 'Internal Server Error' });
    }
  }
  next();
};
```

- **uri**: Retrieves the MongoDB connection string from environment variables.
- **isConnected**: A flag to track whether the database connection is established.
- **connectToMongoDB**: Middleware function to connect to MongoDB if not already connected. It handles connection errors and ensures MongoDB is connected before proceeding to the next middleware or route handler.

Mongoose Schema and Model

```
const movieSchema = new mongoose.Schema({
  title: String,
  genres: [String],
  cast: [String],
  year: Number
}, { collection: 'movies' }); // Set the collection name

const Movie = mongoose.model('Movie', movieSchema);
```

- **movieSchema**: Defines the structure of documents in the movies collection.
- **Movie**: Mongoose model based on the schema, used to interact with the movies collection in MongoDB.

Utility Functions

```
const isValidObjectId = (id) => {
  return mongoose.Types.ObjectId.isValid(id) && (String(new mongoose.Types.ObjectId(id)) === id);
};

const getRandomElements = (array, count) => {
  const shuffled = array.sort(() => 0.5 - Math.random());
  return shuffled.slice(0, count);
};
```

- **isValidObjectId**: Checks if a given ID is a valid MongoDB ObjectId.
- **getRandomElements**: Selects a specified number of random elements from an array.

Route Handlers

Get All Movies

```
app.get('/movies', connectToMongoDB, async (req, res) => {
    try {
        const { page = 1, limit = 500 } = req.query; // Default to
        const movies = await Movie.find()
                                  .skip((page - 1) * limit)
                                  .limit(parseInt(limit));
        res.json(movies);
    } catch (error) {
        console.error('Error retrieving movies:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
```

- /movies: Retrieves a paginated list of movies from the database.

Post Movie IDs

```
app.post('/movies', connectToMongoDB, async (req, res) => {
  const { movie_ids } = req.body;

  try {
      if (!Array.isArray(movie_ids)) {
          return res.status(400).json({ error: 'Invalid input, expected an array of movie IDs' });
      }

      // Validate and convert movie IDs to ObjectId format
      const validObjectIds = movie_ids.filter(id => isValidObjectId(id))
                                      .map(id => new mongoose.Types.ObjectId(id));

      if (validObjectIds.length === 0) {
          return res.status(400).json({ error: 'No valid movie IDs provided' });
      }

      const movies = await Movie.find({ _id: { $in: validObjectIds } });
      res.json(movies);
  } catch (error) {
      console.error('Error retrieving movie details:', error);
      res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

- /movies: Accepts a list of movie IDs and retrieves those movies from the database.

Get Initial Recommendations

```
app.get('/initial-recommendations', connectToMongoDB, async (req, res) => {
  try {
      // Fetch distinct genres
      const genres = await Movie.distinct('genres');

      if (genres.length === 0) {
          return res.status(404).json({ error: 'No genres found' });
      }

      // Randomly select 5 genres
      const selectedGenres = getRandomElements(genres, 5);

      // Fetch 5 movies for each selected genre
      const recommendations = await Promise.all(selectedGenres.map(async (genre) => {
          const movies = await Movie.find({ genres: genre }).limit(5);
          return {
              genre,
              movies
          };
      }));

      res.json(recommendations);
  } catch (error) {
      console.error('Error retrieving initial recommendations:', error);
      res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

- **/initial-recommendations**: Fetches distinct genres, selects a few at random, and retrieves movies for those genres.

Post Similar Movies

```
app.post('/similar', connectToMongoDB, async (req, res) => {
    const { genres, cast } = req.body;

    try {
        // Fetch similar movies based on cast
        const similarActors = await Movie.find({ cast: { $in: cast } }).limit(10);
        // Fetch similar movies based on genres
        const similarGenres = await Movie.find({ genres: { $in: genres } }).limit(10);

        res.json({
            actors: similarActors,
            genres: similarGenres
        });
    } catch (error) {
        console.error('Error retrieving recommendations:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
```

- **/similar**: Retrieves movies similar to those specified based on cast and genres.

Health Check

```
app.get('/', (req, res) => {
    res.json({ message: 'Express.js server is running' });
});
```

- **/**: A simple endpoint to verify that the server is running.

Server Startup

```
if (process.env.NODE_ENV !== 'test') {
  app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
  });
}
```

- **app.listen(port)**: Starts the server on the specified port, unless the environment is set to 'test' (often used in testing environments to avoid starting the server).

Export for Testing

```
export default app;
```

- **export default app**: Exports the Express application instance for use in testing or other modules.

Recommendation Service Machine Learning API Server with FastAPI

Imports and Setup

- **FastAPI**: The main framework for building the API.
- **HTTPException**: Used to raise HTTP errors with specific status codes and messages.
- **BaseModel**: From pydantic, used to define data models with validation.
- **requests**: Used to make HTTP requests.
- **List**: Type hint for lists.
- **os**: For accessing environment variables.
- **load_dotenv**: Loads environment variables from a .env file.
- **CORSMiddleware**: Middleware to handle Cross-Origin Resource Sharing (CORS) requests.
- **numpy**: Used for numerical operations, including sorting.
- **cosine_similarity**: Computes cosine similarity between vectors.
- **CountVectorizer**: Converts text data into a matrix of token counts.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import requests
from typing import List
import os
from dotenv import load_dotenv
from fastapi.middleware.cors import CORSMiddleware
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import CountVectorizer
```

Load Environment Variables

- **load_dotenv()**: Loads environment variables from a .env file into os.environ.

```
load_dotenv()
```

FastAPI Application and CORS Middleware

- **app = FastAPI()**: Creates a new FastAPI application instance.
- **CORSMiddleware**: Configures CORS to allow all origins, methods, and headers. In production, you should specify allowed origins to enhance security.

```
app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Adjust this to your frontend's URL
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Request Models

```
class MovieIdsRequest(BaseModel):
    movieIds: List[str]

class SimilarRequest(BaseModel):
    genres: List[str]
    cast: List[str]
    title: str
```

Fetch Movie Data

- **fetch_movie_data**: Function to make POST requests to the Express.js backend. It handles requests for movie details or similar movies based on provided parameters and handles errors.

```python
# Unified function to fetch movie details or similar movies
def fetch_movie_data(movie_ids: List[str] = None, genres: List[str] = None, cast: List[str] = None):
    try:
        expressjs_base_url = os.getenv("EXPRESSJS_BASE_URL")
        if not expressjs_base_url:
            raise ValueError("Express.js base URL is not set in environment variables")

        if movie_ids:
            url = f"{expressjs_base_url}/movies"
            payload = {"movie_ids": movie_ids}
        elif genres and cast:
            url = f"{expressjs_base_url}/similar"
            payload = {"genres": genres, "cast": cast}
        else:
            raise ValueError("Insufficient parameters provided for request")

        response = requests.post(url, json=payload)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching data from Express.js: {e}")
        return None
    except ValueError as e:
        print(e)
        return None
```

Fetch All Movies

- **fetch_all_movies**: Function to make a GET request to retrieve all movies from the Express.js backend. It handles errors similarly to fetch_movie_data.

```python
# Helper function to fetch all movies for similarity comparison
def fetch_all_movies():
    try:
        expressjs_base_url = os.getenv("EXPRESSJS_BASE_URL")
        if not expressjs_base_url:
            raise ValueError("Express.js base URL is not set in environment variables")

        url = f"{expressjs_base_url}/movies"
        response = requests.get(url)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching all movies from Express.js: {e}")
        return None
    except ValueError as e:
        print(e)
        return None
```

Compute Similarity

- **compute_similarity**: Function to compute similarity scores between selected movies and all movies using cosine similarity. It converts movie features (genres and cast) into vectors and calculates the similarity.

```python
# Helper function to compute similarity
def compute_similarity(selected_movies, all_movies):
    def create_features(movie):
        return " ".join(movie['genres'] + movie['cast'])

    selected_features = [create_features(movie) for movie in selected_movies]
    all_features = [create_features(movie) for movie in all_movies]

    vectorizer = CountVectorizer().fit(all_features)
    selected_vectors = vectorizer.transform(selected_features)
    all_vectors = vectorizer.transform(all_features)

    similarity_scores = cosine_similarity(selected_vectors, all_vectors)
    return similarity_scores.mean(axis=0)
```

Endpoints

Recommend Movies

- **/recommend**: Endpoint to recommend movies based on a list of movie IDs. It retrieves the movies, computes similarity scores, and returns the top 3 recommendations.

```python
@app.post("/recommend")
async def recommend(request: MovieIdsRequest):
    movie_ids = request.movieIds

    selected_movies = fetch_movie_data(movie_ids=movie_ids)
    if not selected_movies:
        raise HTTPException(status_code=500, detail="Error fetching movie details")

    all_movies = fetch_all_movies()
    if not all_movies:
        raise HTTPException(status_code=500, detail="Error fetching all movies")

    similarity_scores = compute_similarity(selected_movies, all_movies)

    selected_movie_titles = {movie['title'] for movie in selected_movies}
    sorted_indices = np.argsort(similarity_scores)[::-1]
    recommendations = [
        all_movies[i] for i in sorted_indices if all_movies[i]['title'] not in selected_movie_titles
    ][:3]  # Top 3 recommendations

    return {"recommendations": recommendations}
```

Similar Movies

- **/similar**: Endpoint to find similar movies based on genres and cast. It filters out the movie with the title specified in the request from the recommendations.

```python
@app.post('/similar')
async def similar(request: SimilarRequest):
    similars = fetch_movie_data(genres=request.genres, cast=request.cast)
    if not similars:
        raise HTTPException(status_code=500, detail="Error fetching similar movies")

    filtered_recommendations = {
        "actors": [movie for movie in similars.get('actors', []) if movie['title'] != request.title],
        "genres": [movie for movie in similars.get('genres', []) if movie['title'] != request.title]
    }

    return filtered_recommendations
```

Health Check

- **/**: A simple endpoint to check if the FastAPI application is running.

```python
@app.get('/')
def read_root():
    return {"message": "FastAPI is running"}
```

Frontend React Native Web

Imports

- **React**: Core React library functions.
- **useState, useEffect**: React hooks for state management and side effects.
- **TextInput, FlatList, Text, View, StyleSheet, ActivityIndicator, Pressable, Alert, ScrollView**: Core React Native components for building the UI.
- **Collapsible**: A third-party component for collapsible sections.
- **API_URL_SEARCH, API_URL_RECOMMEND**: Environment variables for API endpoints.

```javascript
import React, { useState, useEffect } from 'react';
import { TextInput, FlatList, Text, View, StyleSheet, ActivityIndicator, Pressable, Alert, ScrollView } from 'react-native';
import Collapsible from 'react-native-collapsible';
import { API_URL_SEARCH, API_URL_RECOMMEND } from '@env';
```

Component: App

```javascript
const App = () => {
```

State Variables

- **searchTerm**: The term currently entered in the search input.
- **moviesData**: Array holding the fetched movie data.
- **filteredData**: Array of movies filtered based on the search term.
- **loading**: Boolean indicating if data is being loaded initially.
- **loadingMore**: Boolean indicating if more movies are being loaded.
- **selectedMovie**: Currently selected movie for viewing similar movies.
- **similars**: Object containing arrays of similar actors and genres.
- **initialRecommendations**: Array of initial movie recommendations.
- **selectedMovies**: Array of movies selected by the user from initial recommendations.
- **newRecommendations**: Array of recommendations based on selected movies.
- **collapsed**: Boolean for toggling the visibility of collapsible sections.

```
const [searchTerm, setSearchTerm] = useState('');
const [moviesData, setMoviesData] = useState([]);
const [filteredData, setFilteredData] = useState([]);
const [loading, setLoading] = useState(true);
const [loadingMore, setLoadingMore] = useState(false);
const [selectedMovie, setSelectedMovie] = useState(null);
const [similars, setSimilars] = useState({ actors: [], genres: [] });
const [initialRecommendations, setInitialRecommendations] = useState([]);
const [selectedMovies, setSelectedMovies] = useState([]);
const [newRecommendations, setNewRecommendations] = useState([]);
const [collapsed, setCollapsed] = useState(false);
const toggleExpand = () => setCollapsed(!collapsed);
```

Functions

- **fetchMovies(pageNum)**: Fetches movies from the API and updates the moviesData and filteredData states. Handles pagination.

```
const fetchMovies = async (pageNum) => {
    setLoading(true);
    try {
        const response = await fetch(`${API_URL_SEARCH}/movies?page=${pageNum}&limit=500`);
        const data = await response.json();
        const formattedData = data.map(movie => ({
            id: movie._id,
            title: movie.title,
            year: movie.year,
            genres: movie.genres || [], // Ensure genres is an array
            cast: movie.cast || [] // Ensure cast is an array
        }));
        setMoviesData(prevData => [...prevData, ...formattedData]);
        setFilteredData(prevData => [...prevData, ...formattedData]);
        setLoading(false);
        setLoadingMore(false);
    } catch (error) {
        console.error('Error fetching movies:', error);
        setLoading(false);
        setLoadingMore(false);
    }
};
```

- **fetchSimilars(movie)**: Fetches similar movies based on the selected movie's details and updates the similars state.

```
const fetchSimilars = async (movie) => {
    try {
        const response = await fetch(`${API_URL_RECOMMEND}/similar`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({
                movie_id: movie.id,
                genres: movie.genres || [], // Ensure genres is an array
                cast: movie.cast || [], // Ensure cast is an array
                title: movie.title || ""
            })
        });
        const data = await response.json();
        setSimilars(data);
    } catch (error) {
        console.error('Error fetching similar:', error);
    }
};
```

- **fetchInitialRecommendations()**: Fetches initial movie recommendations and updates the initialRecommendations state.

```
const fetchInitialRecommendations = async () => {
    try {
        const response = await fetch(`${API_URL_SEARCH}/initial-recommendations`);
        const data = await response.json();
        setInitialRecommendations(data);
    } catch (error) {
        console.error('Error fetching initial recommendations:', error);
    }
};
```

- **handleSelectMovie(movie)**: Sets the selected movie and fetches similar movies.

```
const handleSelectMovie = (movie) => {
    setSelectedMovie(movie);
    fetchSimilars(movie);
};
```

- **handleSelectInitialMovie(movie)**: Handles the selection of a movie from the initial recommendations, allowing a maximum of 3 selections.

```
const handleSelectInitialMovie = (movie) => {
    const isSelected = selectedMovies.some(selected => selected._id === movie._id);
    if (isSelected) {
        setSelectedMovies(prev => prev.filter(selected => selected._id !== movie._id));
    } else if (selectedMovies.length < 3) {
        setSelectedMovies(prev => [...prev, movie]);
    } else {
        Alert.alert('Limit Reached', 'You can only select up to 3 movies.');
    }
};
```

- **submitSelectedMovies()**: Submits the selected movies to get new recommendations and toggles the collapsible section.

```
const submitSelectedMovies = async () => {
    try {
        const movieIds = selectedMovies.map(movie => movie._id);
        const response = await fetch(`${API_URL_RECOMMEND}/recommend`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ movieIds })
        });
        const data = await response.json();
        setNewRecommendations(data.recommendations);
        setCollapsed(true);
    } catch (error) {
        console.error('Error fetching new recommendations:', error);
    }
};
```

- **loadMoreMovies()**: Loads more movies by fetching the next page of results.

```
const loadMoreMovies = () => {
    setLoadingMore(true);
    fetchMovies(Math.floor(moviesData.length / 10) + 1);
};
```

useEffect Hooks

- **useEffect**: Runs fetchMovies and fetchInitialRecommendations on initial render to load data.

```
useEffect(() => {
    fetchMovies(1); // Load the first page of movies
    fetchInitialRecommendations(); // Fetch initial recommendations
}, []);
```

- **useEffect**: Filters moviesData based on searchTerm and updates filteredData.

```
useEffect(() => {
    const filteredMovies = moviesData.filter(movie =>
        movie.title.toLowerCase().includes(searchTerm.toLowerCase())
    );
    setFilteredData(filteredMovies);
}, [searchTerm, moviesData]);
```

Render Logic

- **Loading State**: Shows an ActivityIndicator if data is loading and no initial recommendations are available.

```
if (loading && !initialRecommendations.length) {
    return (
        <View style={styles.container}>
            <ActivityIndicator size="large" color="#0000ff" />
        </View>
    );
}
```

- **Collapsible Section**: Contains initial recommendations where users can select up to 3 movies. Displays a button to expand or collapse this section.

```
return (
    <View style={styles.container}>
        <Text style={styles.title}>Movie Recommender and Search Application</Text>
        <ScrollView contentContainerStyle={styles.scrollContainer}>
            {/* Collapsible Section */}
            <View style={styles.collapsibleContainer}>
                <Pressable onPress={toggleExpand} style={styles.button}>
                    <Text style={styles.buttonText}>{collapsed ? 'Pick 3 Movies' : 'Hide'}</Text>
                </Pressable>
                <Collapsible collapsed={collapsed}>
```

```
{/* Initial Recommendations */}
<View style={styles.initialRecommendationsContainer}>
    <Text style={styles.title}>Initial Recommendations - Pick 3 Movies</Text>
    <FlatList
        data={initialRecommendations}
        keyExtractor={getUniqueKey}
        renderItem={({ item }) => (
            <View style={styles.card}>
                <Text style={styles.itemTitle}>{item.genre}</Text>
                <FlatList
                    data={item.movies}
                    keyExtractor={getUniqueKey}
                    renderItem={({ item }) => (
                        <Pressable onPress={() => handleSelectInitialMovie(item)}>
                            <View style={[styles.card, selectedMovies.some(selected => selected._id === item._id) && styles.selectedCard]}>
                                <Text style={styles.itemTitle}>{item.title}</Text>
                            </View>
                        </Pressable>
                    )}
                    horizontal
                />
            </View>
        )}
        horizontal
    />
    <Pressable
        style={[styles.button, { opacity: selectedMovies.length < 3 ? 0.5 : 1 }]}
        onPress={submitSelectedMovies}
        disabled={selectedMovies.length < 3}
    >
        <Text style={styles.buttonText}>Submit Selected Movies</Text>
    </Pressable>
</View>
```

- **Similar Movies**: Shows similar movies based on the selected movie, categorized by similar actors and genres.

```
{/* Similar Movies */}
{selectedMovie && (
    <View style={styles.similarsContainer}>
        <Text style={styles.title}>Similar Casts</Text>
        <FlatList
            horizontal
            data={similars.actors}
            keyExtractor={getUniqueKey}
            renderItem={({ item }) => (
                <View style={styles.card}>
                    <Text style={styles.itemTitle}>{item.title}</Text>
                    <Text style={styles.itemText}>Year: {item.year}</Text>
                    <Text style={styles.itemText}>Genres: {item.genres?.join(', ') || 'N/A'}</Text>
                    <Text style={styles.itemText}>Cast: {item.cast?.join(', ') || 'N/A'}</Text>
                </View>
            )}
        />
        <Text style={styles.title}>Similar Genres</Text>
        <FlatList
            horizontal
            data={similars.genres}
            keyExtractor={getUniqueKey}
            renderItem={({ item }) => (
                <View style={styles.card}>
                    <Text style={styles.itemTitle}>{item.title}</Text>
                    <Text style={styles.itemText}>Year: {item.year}</Text>
                    <Text style={styles.itemText}>Genres: {item.genres?.join(', ') || 'N/A'}</Text>
                    <Text style={styles.itemText}>Cast: {item.cast?.join(', ') || 'N/A'}</Text>
                </View>
            )}
        />
    </View>
)}
```

- **New Recommendations**: Displays new recommendations based on the selected movies and allows the user to view similar cast and genres.

```
{/* New Recommendations */}
{newRecommendations.length > 0 && (
    <View style={styles.newRecommendationsContainer}>
        <Text style={styles.title}>New Recommendations - Select any movie to get similar cast and genres</Text>
        <FlatList
            data={newRecommendations}
            keyExtractor={getUniqueKey}
            renderItem={({ item }) => (
                <Pressable onPress={() => handleSelectMovie(item)}>
                    <View style={styles.card}>
                        <Text style={styles.itemTitle}>{item.title}</Text>
                        <Text style={styles.itemText}>Year: {item.year}</Text>
                        <Text style={styles.itemText}>Genres: {item.genres?.join(', ') || 'N/A'}</Text>
                        <Text style={styles.itemText}>Cast: {item.cast?.join(', ') || 'N/A'}</Text>
                    </View>
                </Pressable>
            )}
        />
    </View>
)}
```

- **Search Functionality**: Allows users to search for movies and load more results. Shows search results and a button to load more movies.

```
{/* Movie Search */}
<View style={styles.searchContainer}>
    <Text style={styles.title}>Search for Movies - Select any movie to get similar cast and genres</Text>
    <TextInput
        style={styles.searchInput}
        placeholder="Search movies..."
        value={searchTerm}
        onChangeText={setSearchTerm}
    />
    <Pressable onPress={loadMoreMovies} style={styles.button}>
        <Text style={styles.buttonText}>Load More Movies</Text>
    </Pressable>
    <FlatList
        data={filteredData}
        keyExtractor={getUniqueKey}
        renderItem={({ item }) => (
            <Pressable onPress={() => handleSelectMovie(item)}>
                <View style={styles.listItem}>
                    <Text style={styles.itemTitle}>{item.title}</Text>
                    <Text style={styles.itemText}>Year: {item.year}</Text>
                    <Text style={styles.itemText}>Genres: {item.genres?.join(', ') || 'N/A'}</Text>
                    <Text style={styles.itemText}>Cast: {item.cast?.join(', ') || 'N/A'}</Text>
                </View>
            </Pressable>
        )}
        ListFooterComponent={() => (
            loadingMore ? <ActivityIndicator size="small" color="#0000ff" /> :
            <Pressable onPress={loadMoreMovies} style={styles.button}>
                <Text style={styles.buttonText}>Load More Movies</Text>
            </Pressable>
        )}
    />
</View>
```

- **Container**: Basic layout styling for the main view.
  Styles

```
const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        backgroundColor: '#fff',
    },
```

- **Scroll Container**: Ensures the scrollable content fills the screen.

```
scrollContainer: {
    flexGrow: 1,
},
```

- **Search Container**: Styles for the search input area.

```
searchContainer: {
    marginTop: 5,
    borderWidth: 2,
    borderColor: '#000',
    borderRadius: 8,
    padding: 5,
    marginBottom: 5,
},
```

- **Card, Selected Card**: Styling for movie items, with selected items highlighted.

```
card: {
    backgroundColor: '#f9f9f9',
    padding: 5,
    marginBottom: 10,
    borderRadius: 5,
},
selectedCard: {
    borderWidth: 2,
    borderColor: 'blue',
},
```

- **Button, Button Text**: Styling for buttons.

```
button: {
    backgroundColor: '#007bff',
    padding: 10,
    borderRadius: 5,
    alignItems: 'center',
},
buttonText: {
    color: '#fff',
    fontSize: 16,
},
```

3) Setup

Backend ExpressJS (under the 'backend-expressjs' directory)

1. Copy your MongoDB Connection String from MongoDB Atlas.

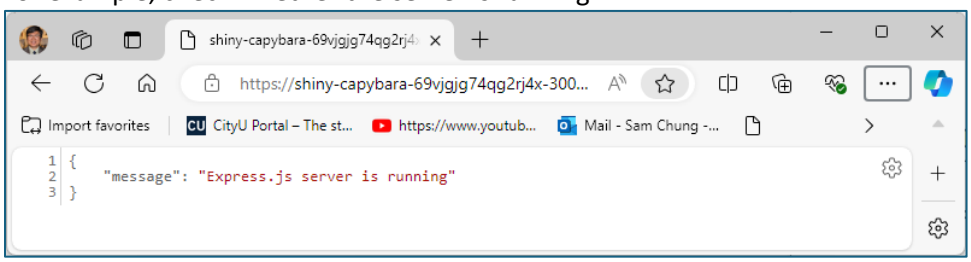
2. Create **.env** file and paste it after **MONGODB_URI=**

```
backend-expressjs > .env
1   MONGODB_URI=mongodb+srv://clarkngo:[redacted]@cluster0.rhkmsrr.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0
```

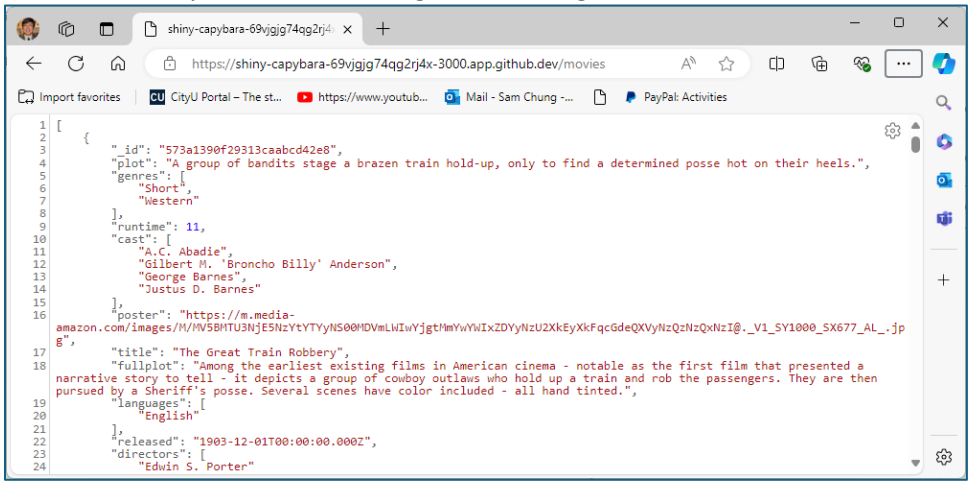
3. Open New Terminal and type the following:
   > cd backend-expressjs
   > npm install
   > npm run start

4. Make the Port Visibility `Public`.

5. Copy the forwarded address of ExpressJS.
   For example, check whether the server is running.



   Check whether your server is taking to the MongoDB Atlas.



6. Test the **/initial-recommendations** endpoint.

7. Have the screenshot of your ‘Initial Recommendations’ as ‘firstname_lastname_initial_recommendations.png’.

4) **Backend FastAPI** (under the 'backend-fastapi' directory)

1. Create `.env` file and paste the forwarded address ExpressJS after `EXPRESSJS_BASE_URL=`

```
backend-fastapi > .env
1   EXPRESSJS_BASE_URL=https://animated-fishstick-r79q5r4475v24xp-3000.app.github.dev
2
```

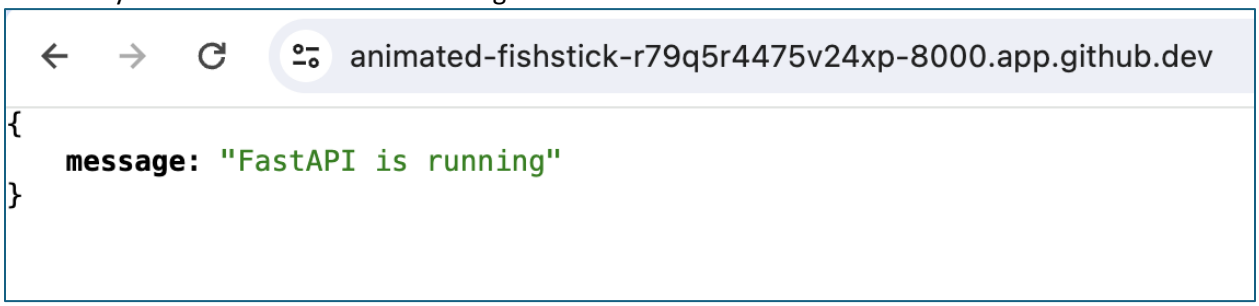
   Caution: Do not have the "/" at the end of the URL.

2. Open New Terminal and type the following:
   > cd backend-fastapi
   > pip install -r requirements.txt
   > uvicorn main:app --host 0.0.0.0 --port 8000

3. Make the Port Visibility `Public`.

4. Copy the forwarded address of FastAPI.

5. Test your FastAPI server if it's running.



5) **Frontend React Native (Web)** (under the 'frontend-reactnative' directory)

1. Create `.env` file and paste the forwarded address ExpressJS after **API_URL_SEARCH=** and forwarded address of FastAPI after **API_URL_RECOMMEND=**

```
frontend-reactnative > .env
1   API_URL_SEARCH=https://animated-fishstick-r79q5r4475v24xp-3000.app.github.dev
2   API_URL_RECOMMEND=https://animated-fishstick-r79q5r4475v24xp-8000.app.github.dev
```

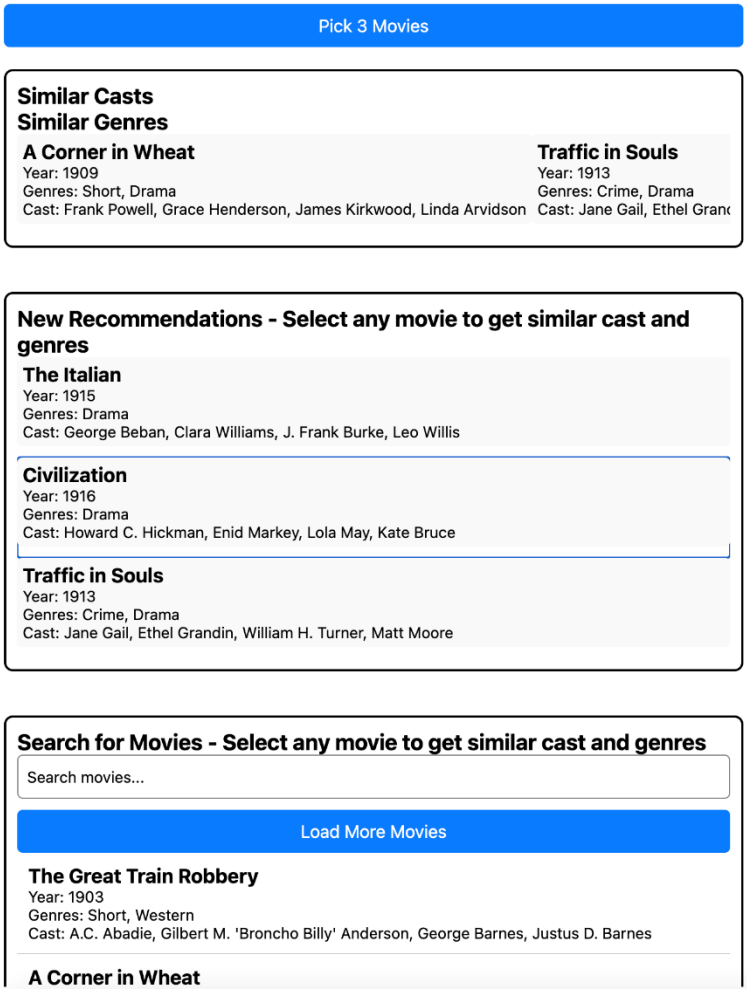
2. Have the screenshot of your 'API URLS' as 'firstname_lastname_api_urls.png'.

3. Open New Terminal
   > cd frontend-reactnative
   > npm install
   > npx expo login
   > npx expo start **--web**

   You may also open in your mobile with
   > npx expo start --tunnel

4. Open your Browser.
5. Select 3 Movies and Submit Selected Movies.

6. Select any movie from New Recommendations or Movie Search to generate similar cast or similar genres.



7. Have a screenshot of your ‘Movie Recommendation’ as ‘firstname_lastname_movie_recommendation.png’.

8. Test out the Load More Movies button
   Note: we have incorporated pagination to load a few movies at a time. This is a tradeoff for less data for more speed. Clicking Load More, will keep fetching more movies in the database.

6) Pushing your work to GitHub

1. Go to Source Control on your GitHub codespaces and observe the pending changes.

2. Type the Message for your changes in the Message box on the top. For example," **Submission for TT06 – Your Name**"

3. Click on the dropdown beside the commit button and select **Commit & Push** to update the changes to your repository main branch.

4. Select **Yes** when prompted.