Bulls Movies

Created by Pedro Bautista, Christiana Hellenbrand & Christopher Ragland

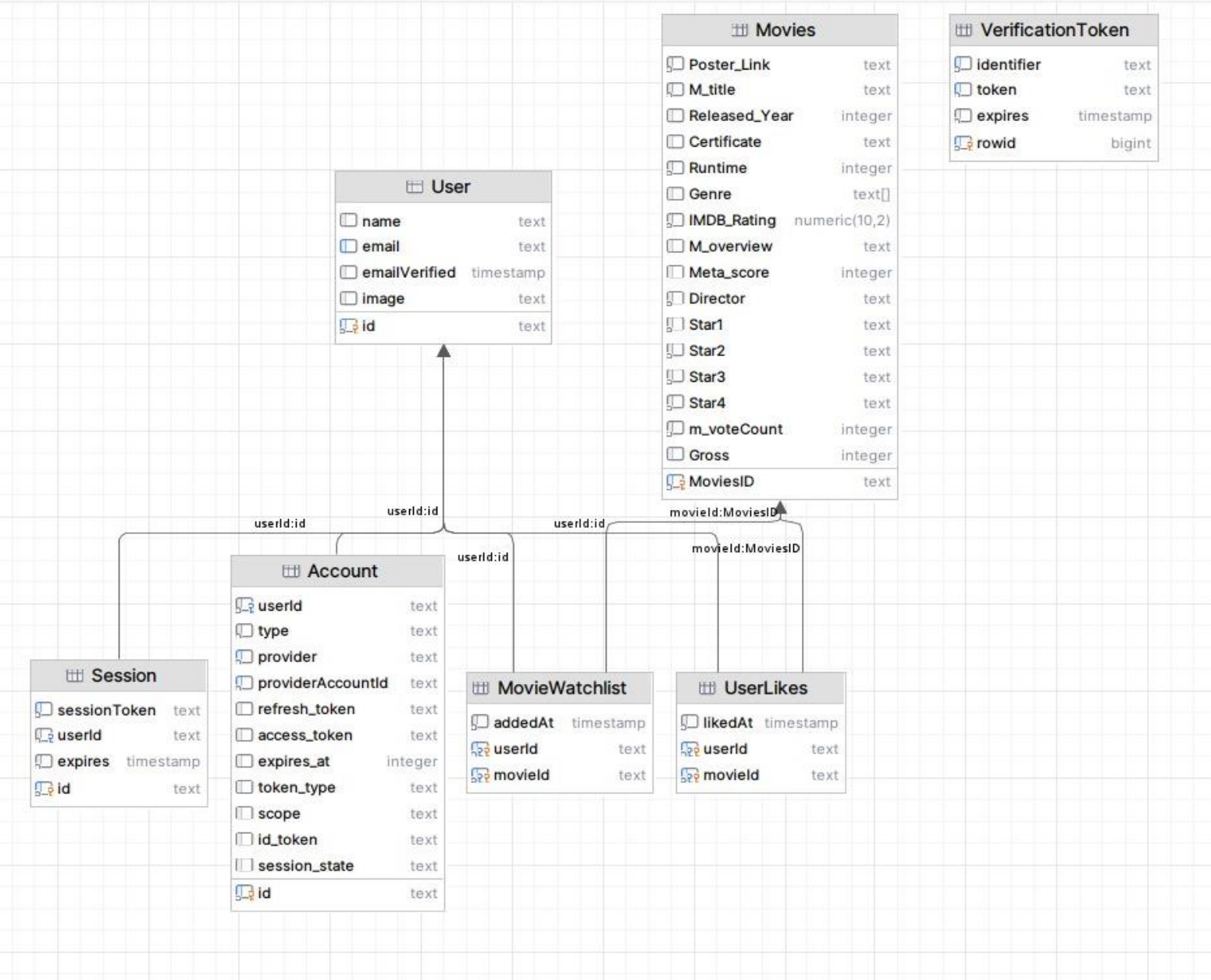
Table of Contents

Overview

Bulls Movies is a web-based application that allows the user to access a database of movie information collected from IMDB. The purpose of this website is to create a user-friendly method to find movies, access favorites and store new movies to a user's watchlist to watch at a later time. The user can log in with either a Google or GitHub account to access a thousand movie cards. Each movie card contains information, such as title, genre, runtime, ratings, director, stars, and overview.

Once logged in, the user is redirected to the Newest Page which includes the top 12 newest/highest rated movies for the user to look through. On the left panel, the Search tool is used for accessing movies by a specific title, genre, year, and rating. Any movie the user likes can be stored on their Favorites Page. The user can add or remove movies to their Watchlist to see the movies they plan to watch in the future.

Entity-Relationship Diagram



Relations:

- One-to-One: User has one Account and one Session
- One-to-Many: UserLikes contains multiple Movies & Watchlist contains multiple Movies

Three-tier Implementation

A. Interface

a. Bulls Movies is implemented through a web user interface created with React for the client UI and actions/procedures are executed through NodeJS via API routes that allows the user to interact with various features on a remote server. The database is connected through the interface via an ORM called Prisma, that calls a Prisma Client to connect with the database using Cockroach DB over the cloud.

B. Application Logic

a. Searching
    i. A form is present on the search page for users to fill out. Using client-side scripting, when the user presses the submit ("Search") button the form values are passed to the API call and within the API call, only information that is not null-ish or undefined is passed to the SQL query. This allows users to enter multiple search parameters simultaneously.

b. Pagination for search page
    i. Instead of repeatedly making a call to the database to return the next 12 or previous 12 movies from the Movies table, the movies that match the user search are returned and stored in an array via JavaScript. This improves the speed and the user experience of the application. When the user clicks on next or previous, the current 12 is either shifted forward 12 or backwards 12 given the user is not at the front of the list or the back of the list.

c. Watchlist
    i. The idea behind Watchlist is, while the user is looking through different movies and they see a movie they want to watch they click the "+" to add the specific movie to the list of movies they plan on watching at a later point in time.

d. Favorites

i. The idea behind the Favorites page is when the user has watched a movie previously and likes the movie; the user can click on the heart button to add/remove it from their favorites page.

C. Database

a. The database was created using SQL on Postgres.

b. Tables

i. Movies

This table stores the information associated with a given movie. When a user searches for a movie or accesses the Learn More page, the Movie table is called to access details of the movie. Each movie is associated to a unique *MoviesID.*

Movies(*MoviesID*: STRING, *Poster_Link*: STRING, *M_title:STRING, Released_Year*: INTEGER, *Certification*: STRING, *Runtime*: INTEGER, *Genre*: STRING[], *IMBD_Rating*: DECIMAL, *M_overview*: STRING, *Meta_score*: INTEGER, *Director*: STRING, *Star1*: STRING, *Star2*: STRING, *Star3*: STRING, *Star4*: STRING, *m_voteCount*: INTEGER, *Gross*: INTEGER)

Foreign keys: none
Candidate key: *MoviesID*
Primary key: *MoviesID*

Not NULL: *MoviesID, Poster_Link,M_title,Runtime, IMBD_Rating, Director, Star1, Star2, Star3, Star4, m_voteCount*

ii. UserLikes

This table stores the movies that a user likes. When the user clicks the heart icon at the top right corner of the movie card, the *movieId* is added to this table with the associated *userId* and when they liked the movie. The user can add or remove a like by clicking the heart icon consecutively.

UserLikes(*userId*: STRING, *movieId*: STRING, *likesAt*: DATETIME)

Foreign keys: *userId, movieId*

Candidate key: *userId, movieId*

Primary key: *[userId , movieId]*

Not NULL: *userId, movieId, likesAt*

iii. MovieWatchlist

This table stores the user's watchlist. The watchlist is a collection of movies that the user plans to watch in the future. The user can add or remove movies from their Watchlist using '+' and '-' at the top right corner of the movie card.

MovieWatchlist(*userId*: STRING, *movieId*: STRING, *addedAt*: DATETIME)

Foreign keys: *userId, movieId*

Candidate key: *userId, movieId*

Primary key: *[userId , movieId]*

Not NULL: *userId, movieId, addedAt*

iv. Account

This table stores the user's account information. When a new user creates an account, their account information is stored here. This keeps track of the number of users using the Bulls Movies system.

Account(*id*: STRING, *userId*: STRING, *type*: STRING, *provider*: STRING, *providerAccountId*: STRING, *refresh_token*: STRING, *access_token*: STRING, *expires_at*: INTEGER, *token_type*: STRING, *scope*: STRING, *id_token*: STRING, *session_state*: STRING)

Foreign keys: *userId*

Candidate key: *id, userId, [provider, providerAccountId]*

Primary key: *id*

Not NULL: *id, userId, [provider, providerAccountId]*

v. Session

This table stores each session that the user accesses Bulls Movies. Each session is associated with a *userId* and the time in which the session token will expire.

Session(*id*: STRING, *sessionToken*: STRING, *userId*: STRING, *expires*: DATETIME)

Foreign keys: *userId*

Candidate key: *id, sessionToken*

Primary key: *id*

Not NULL: *id, sessionToken, userId,expires*

vi. User

This table stores the user's account, session, likes, and watchlist information. *userId* is used as a foreign key to associate the user with additional attributes in other tables.

User(*id*: STRING, *name*: STRING, *email*: STRING. *emailVerified*: DATETIME, *image*: STRING, *accounts*: ACCOUNT[], *sessions*: SESSION[], *UserLikes*: USERLIKES[], *MovieWatchlist*: MOVIEWATCHLIST[])

Foreign keys: *accounts, sessions, UserLikes, MovieWatchlist*

Candidate key: *id, email*

Primary key: *id*

Not NULL: *id, email, accounts, sessions, UserLikes, MovieWatchlist*

vii. VerificationToken

This table holds the token-based authentication for the user to verify their identity. When accessing their account, they receive a unique token that gives them access and expires after a certain amount of time.

VerificationToken(*identifier*: STRING, *token*: STRING, *expires*: DATETIME)

Foreign keys: none

Candidate key: *identifier, token*

Primary key: *[identifier, token]*

Not NULL: *identifier, token, expires*

c. Queries

i. User Sign-In

When the user clicks the login button, the database is called, and a new user is added to the Account table. While in the database, the user information is either created or collected and a session token is created that will give the user access for the duration of the token's life span.

```
14          <button
15            onClick={() => signIn(undefined, {callbackUrl: "/recently-added"})}
16            className="text-white px-3 py-2 bg-teal-600 rounded-md shadow-sm hover:bg-teal-700"
17            >
18            Login / Sign up
19          </button>
```

```
9   const prisma = new PrismaClient()
10
11  export const authOptions = {
12      adapter: PrismaAdapter(prisma),
13      providers: [
14          GitHubProvider({
15                  clientId: process.env.GITHUB_CLIENT_ID,
16                  clientSecret: process.env.GITHUB_CLIENT_SECRET,
17          }),
18          GoogleProvider({
19              clientId: process.env.GOOGLE_CLIENT_ID,
20              clientSecret: process.env.GOOGLE_CLIENT_SECRET,
21          }),
22      ],
23      secret: process.env.NEXTAUTH_SECRET,
24  };
```

ii. Recently Added

First, the authenticity of the user is checked. If the user is not logged in, they are redirected to the login page. Else, an async call is made to the database. Lastly, the top twelve new movies are called from the database and returned to the Newest page.

```
export async function getServerSideProps(context)
{
  //if user is not logged in
  //redirect to login
  const session = await getSession(context);
  if (!session)
  {
    return {
      redirect: {
        destination: "/",
        permanent: false,
      }
    }
  }

  const movies = await prisma.movies.findMany({
    orderBy: [
      { Released_Year: 'desc' },
      { IMDB_Rating: 'desc' },
    ],
    take: 12
  });
  return {
    props: {
      movies: JSON.parse(JSON.stringify(movies)),
    }
  }
}
```

iii. Search

The user fills out the search criteria based on four categories: The movie's title (case sensitive), the year the movie was released, the genre of the movie, and the rating of the movie. The results are ordered by title in alphabetical order. A count of the total movies that match the criteria is collected and returned as well.

```
import prisma from "../../../lib/prisma";

export default async function handler(req, res)
{
  try {
    const { title, year, rating, genre } = req.body;
    const where = {};

    if (title != "")
    {
      where.M_title = { contains: title };
    }

    if (year != "")
    {
      where.Released_Year = parseInt(year);
    }

    if (rating != 1.0)
    {
      where.IMDB_Rating = { gte : rating };
    }

    if (genre != "")
    {
      where.Genre = { has : genre };
    }

    const movies = await prisma.movies.findMany({
      where,
      orderBy: {
        M_title: "asc"
      },

    })

    const movieCount = await prisma.movies.count({
      where,
    })

    return res.status(200).json({ movies, movieCount });
  }
  catch (error)
  {
    console.error(error);

    return res.status(500).json({error: "Failed to get movies"});
  }
}
```

iv. Watchlist

When the user accesses the watchlist page, user authenticity is first checked before prompting the query. If not, user is redirected to login page. Else, user email is used to go to Watchlist table and retrieve all the movies associated with user email.

```
export async function getServerSideProps(context)
{
  //if user is not logged in
  //redirect to login
  const session = await getSession(context);
  if (!session)
  {
    return {
      redirect: {
        destination: "/",
        permanent: false,
      }
    }
  }
  //get user information
  const user = await prisma.user.findUnique({
    where : {
      email: session.user.email
    }
  })
  const userId = user.id;
  //get watchlist information
  const moviesFromWatchlist = await prisma.movieWatchlist.findMany({
    where: {
      userId: userId,
    },
  })

  // Get movie information
  const movieIds = moviesFromWatchlist.map((movie) => movie.movieId)
  const movies = await prisma.movies.findMany({
    where: {
      MoviesID: {
        in: movieIds,
      },
    },
  })

  return {
    props: {
      movies: JSON.parse(JSON.stringify(movies)),
    }
  }
}
```

v. Favorites

When the user accesses the favorites page, user authenticity is first checked before prompting the query. If not, the user is redirected to the login page. Else, the user email is used to return all the movies associated with the user within the UserLikes table.

```
export async function getServerSideProps(context)
{
  //if user is not logged in
  //redirect to login
  const session = await getSession(context);
  if (!session)
  {
    return {
      redirect: {
        destination: "/",
        permanent: false,
      }
    }
  }
  //get user information
  const user = await prisma.user.findUnique({
    where : {
      email: session.user.email
    }
  })
  const userId = user.id;
  //get watchlist information
  const moviesFromFavorites = await prisma.userLikes.findMany({
    where: {
      userId: userId,
    },
  })
  // Get movie information
  const movieIds = moviesFromFavorites.map((movie) => movie.movieId)
  const movies = await prisma.movies.findMany({
    where: {
      MoviesID: {
        in: movieIds,
      },
    },
  })
  return {
    props: {
      movies: JSON.parse(JSON.stringify(movies)),
    }
```

vi. Add to Watchlist/Favorites

A user clicks the '+' on the movie card to add to Watchlist and the heart icon to add to Favorites. The specific card's *movieId* as well as the user's email is passed to an API call that is sent to the MovieWatchlist table or the Favorites table. A record is created with the *movieId* and *userId* and is added to the Watchlist/Favorites.

```
export default async function handler(req, res)
{
  try
  {
    const { email, movieId } = await req.body;

    //get user info
    const user = await prisma.user.findUnique({
      where : {
        email: email
      }
    })

    const userId = user.id;

    if (movieId)
    {
      await prisma.userLikes.create({
        data: {
          userId,
          movieId,
        }
      })
    }

```

```
export default async function handler(req, res)
{
  try
  {
    const { email, movieId } = await req.body;
    //get user info
    const user = await prisma.user.findUnique({
      where : {
        email: email
      }
    })
    const userId = user.id;
    if (movieId)
    {
      await prisma.movieWatchlist.create({
        data: {
          userId,
          movieId,
        }
      })
    }
    res.status(200).json({message: "Successfully added to watchlist"});
  }
```

vii. Remove from Watchlist/Favorites

A movie is present in the user's Watchlist and/or user's Favorites. The user clicks "-" or the heart icon on the movie card which triggers an API call passing the user's email and the specific *movieId* for that card. The user's email is queried in the User table to find the *userId* which is then used with the MovieWatchlist or Favorites table to find the matching record. The matching record is then removed from the table.

```
export default async function handler(req, res)
{
  try
  {
    const { email, movieId } = await req.body;
    //get user info
    const user = await prisma.user.findUnique({
      where : {
        email: email
      }
    })
    const userId = user.id;
    //delete the associated element from the table
    await prisma.movieWatchlist.delete({
      where: {
        userId_movieId: { userId, movieId }
      }
    })
    res.status(200).json({message: "Successfully removed from watchlist"});
  }
  catch (error)
  {
    console.error(error);
    res.status(500).json({error: "Failed to remove from watchlist"})
  }
}
```

```
export default async function handler(req, res)
{
  try
  {
    const { email, movieId } = await req.body;

    //get user info
    const user = await prisma.user.findUnique({
      where : {
        email: email
      }
    })

    const userId = user.id;

    await prisma.userLikes.delete({
      where: {
        userId_movieId: { userId, movieId }
      }
    })

    res.status(200).json({message: "Successfully removed from favorites"});
  }
  catch (error)
  {
    console.error(error);
    res.status(500).json({ error: "Failed to remove favorites"})
  }
}
```

viii. Learn More

If the user is not signed in, they are redirected to login page. Otherwise, the *movieId* that passed to the URL is found in the database and returns information pertaining to the movie. This includes title, genre, overview, director, runtime, IMDB rating, movie's stars, and release year.

```
export async function getServerSideProps(context)
{
  //if user is not logged in
  //redirect to login
  const session = await getSession(context);
  if (!session)
  {
    return {
      redirect: {
        destination: "/",
        permanent: false,
      }
    }
  }
  const { movieId } = context.query;
  const movie = await prisma.movies.findUnique({
    where: {
      MoviesID: movieId,
    },
  });
  return {
    props: { movie : JSON.parse(JSON.stringify(movie)) }
  };
}
```