# Exercises: ASP.NET Core - Identity

Problems for exercises for the ["ASP.NET Core Fundamentals" course @ SoftUni](https://softuni.bg/trainings/4707/asp-net-fundamentals-september-2024)

A movie ticket and popcorn

Description automatically generated

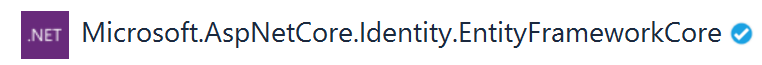
## CinemaWebApp – Authentication & Authorization

In this project, we will implement **ASP.NET Core Identity** to manage **authentication** and **authorization** within our application. Authentication ensures that users can securely **register**, **log in**, and **access** the application, while authorization will determine what authenticated users are allowed to do within the system. We will be using the **default Identity system**, which provides out-of-the-box functionality for handling user accounts, password management, and secure access to protected resources. The focus of this lesson will be on **enabling authentication** and **securing specific parts of the application**. If time permits, we may also introduce **user roles** to demonstrate how to further control access based on user types.

### Install Required Packages

In order to use Identity in our project, we need to install the required packages. These packages provide the core functionality for handling user authentication and authorization. To install them, follow these steps:

* Open **NuGet Package Manager** (right-click on the project in **Solution Explorer** > Manage NuGet Packages)
* Search for and install the following packages:
  + **Microsoft.AspNetCore.Identity.EntityFrameworkCore**

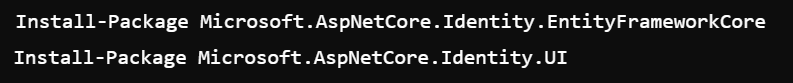
****

* + **Microsoft.AspNetCore.Identity.UI**

**A black and blue text

Description automatically generated**

You can also install them using the Package Manager Console with the following commands:



Once installed, these packages will allow us to add Identity features to our application.

### Configure Identity in Program.cs

We now need to configure **ASP.NET Core Identity** in the Program.cs file. This will integrate user authentication into our project using the default Identity system.

* Open the **Program.cs** file and add the following code to **configure Identity services**:

A close up of words

Description automatically generated

* + This sets up Identity to use **Entity Framework** for storing user information and configures the default user model (IdentityUser)
* Ensure that **authentication** and **authorization** middleware are added

A close-up of a white background

Description automatically generated

* Lastly, make sure the **Identity UI is mapped** by adding



Once configured, this will allow us to use **Identity** for **managing user authentication** in the application.

### Integrate Identity within existing DbContext

Now that we have a DbContext managing our models (Movie, Cinema, etc.), we need to integrate **ASP.NET Core Identity** to handle user authentication.

This involves modifying our existing **AppDbContext** so it can handle both our models and the Identity system.

* Open the **AppDbContext** class and modify it to inherit from **IdentityDbContext<IdentityUser>**
* This change will allow the **AppDbContext** to manage both the Identity tables (for authentication) and your existing entities (Movie, Cinema, CinemaMovie)
* The rest of your configurations for composite keys and relationships will remain the same

A close-up of a white background

Description automatically generated

#### Create and Apply Migration

Our **AppDbContext** is configured to handle both user authentication (via Identity) and our existing models, we need to create a migration to update the database schema.

* Open the **Package Manager Console** and run the following command to create a migration:

A black and white text

Description automatically generated

* This command generates the necessary migration files to reflect the changes in the database schema

A screenshot of a computer

Description automatically generated

* Once the migration is created, **apply the migration to the database**



* This command updates the database, adding the Identity tables along with the existing tables for **Movie**, **Cinema**, and **CinemaMovie**. After this step, your database will be ready to handle user authentication.

A screenshot of a computer

Description automatically generated

### Add Authentication UI

By default, **ASP.NET Core Identity** provides built-in **Razor Pages** for user **registration**, **login**, and   
**password management**. We’ll use these built-in pages to quickly enable authentication in the project.

#### Scaffold Identity through Visual Studio UI

To scaffold the **Identity UI** through **Visual Studio**, follow these steps:

* **Right-click on your project** in the **Solution Explorer**
* Select **Add** > **New Scaffolded Item**
* In the **Add Scaffold** dialog, choose **Identity**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

* In the **Identity Scaffolding** dialog, select the pages you want to scaffold (e.g., Login, Register, Logout)
* Select **ApplicationDbContext** (or your Identity database context) in the **Data Context Class** dropdown
* Click **Add** to scaffold the pages

A screenshot of a computer

Description automatically generated

* This process will add the necessary Identity pages (like login, register) to the project. You can customize them as needed

#### Understanding the Impact of Scaffolding Identity

After scaffolding Identity into your project, several key changes have been made. Let's explore what has been added and how it impacts the project

* **New Files** in **Areas/Identity/Pages**
  + A new folder structure has been added to your project: **Areas/Identity/Pages/Account**
  + This folder contains key pages for user authentication

A screenshot of a computer

Description automatically generated

* **This folder contains** key pages for user authentication
  + **Login.cshtml** and **Login.cshtml.cs**
  + **Register.cshtml** and **Register.cshtml.cs**
  + **Logout.cshtml** and others
* **Code-Behind Files**
  + Each Identity page has an associated **code-behind file** (e.g., Login.cshtml.cs) where the logic for user authentication is handled. You can view and customize these files to modify how data is processed
* **Changes** to **\_Layout.cshtml**
  + You'll need to add **Login**, **Register**, and **Logout** links to the navigation bar in the **\_Layout.cshtml** file if not already present. These links will now point to the scaffolded Identity pages

A screenshot of a computer

Description automatically generated

* **New Identity Routes** - The scaffolded Identity pages are now accessible via routes like:
  + **/Identity/Account/Login**
  + **/Identity/Account/Register**
  + **/Identity/Account/Logout**
* **Authentication**
  + After scaffolding, your **application is fully equipped to handle user authentication with Identity**.
  + **Users can log in, register, and log out**

**A screenshot of a computer

Description automatically generated**

#### Why Use a Partial View?

Using a partial view like **\_LoginPartial** helps in keeping the authentication logic separated and **reusable across multiple views**. Instead of writing login/logout links multiple times, we just reference the partial, and it automatically handles the user’s login status.

Once added, the navbar will dynamically show the correct authentication links based on the user's login state.

#### Adding the Login Partial in the Navbar

To manage the authentication links (Login, Register, and Logout), we use a **partial view** called **\_LoginPartial.cshtml**. This partial dynamically displays the correct links based on whether the user is logged in or not.

* Open the **\_Layout.cshtml** file in the **Views/Shared** folder
* Add the following line where you want the login, register, and logout links to appear:

A screenshot of a computer code

Description automatically generated

This partial:

* Displays the **Login** and **Register** links when the user is **not logged in**
* Displays the **Logout** link and a greeting (e.g., "Hello, [Username]") when the user is **logged in**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

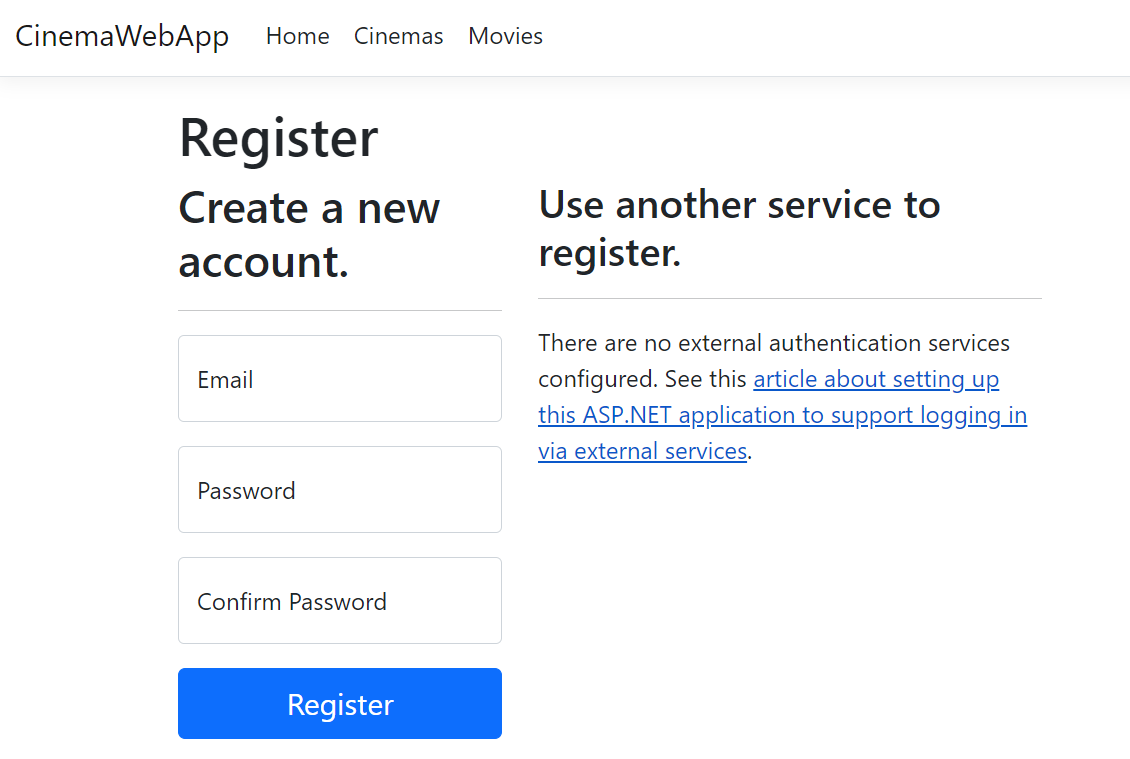
Description automatically generated



A screenshot of a computer

Description automatically generated

#### Simplifying the Registration Page

As you can see, the default **Register** page provided by ASP.NET Identity includes several options and features, such as external logins (Google, Facebook, etc.), and a range of configurations for different scenarios. However, for the purpose of this lesson and to keep things simple, we will focus on the essential elements needed for user registration.

We will keep only the following fields:

* **Email**: This is the user’s unique identifier for creating an account
* **Password**: The password that the user will use to log in
* **Confirm Password**: To ensure the password is entered correctly, users must confirm it by typing it twice

We are excluding external login options and other advanced features. These can be added later if necessary.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

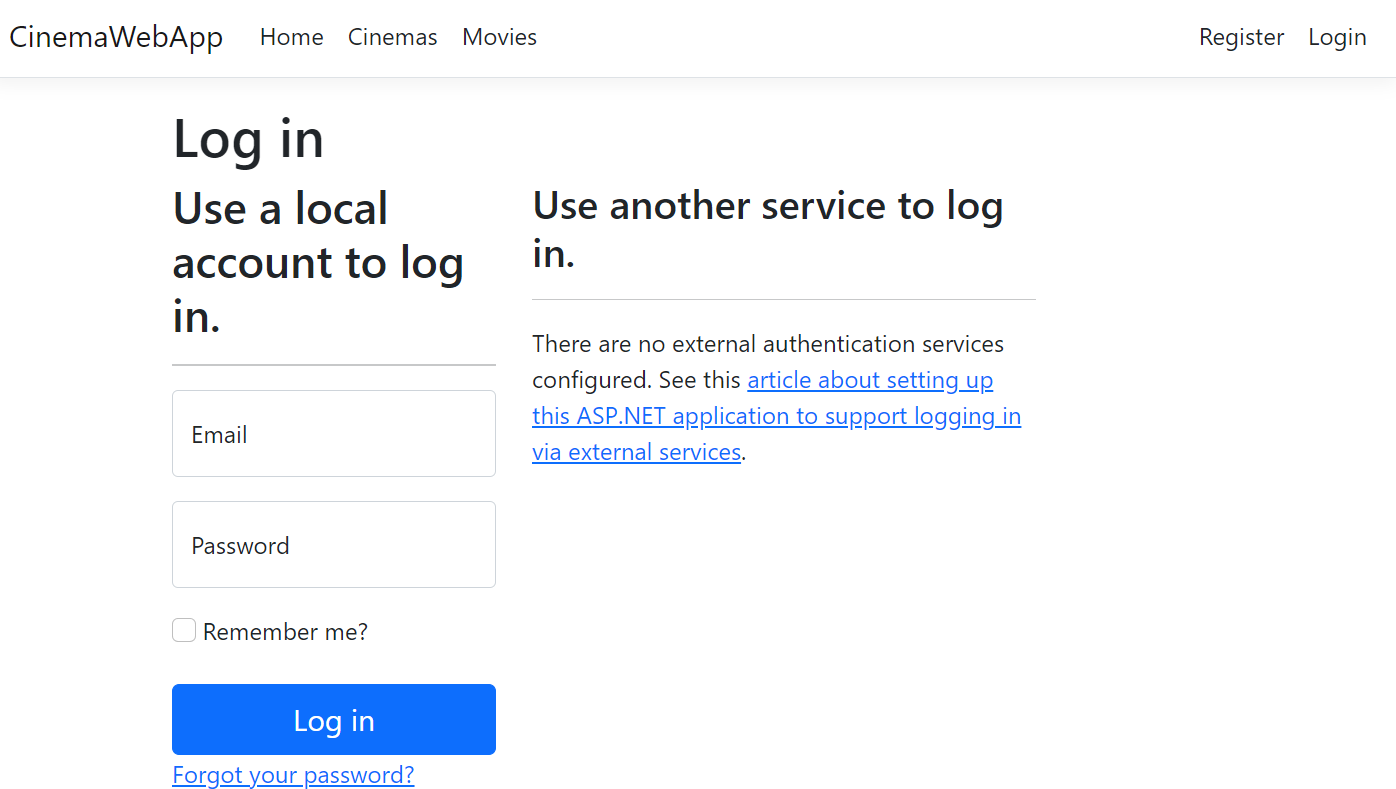
Description automatically generated

#### Simplifying the Login Page

Just like the **Register** page, the default **Login** page provided by ASP.NET Identity comes with several advanced features, such as external login options (logging in with Google, Facebook, etc.), and additional configuration options. For now, we will stick with the essential fields needed for a basic login experience.

We will keep the following fields:

* **Email**: The email address the user registered with
* **Password**: The password that the user created during registration



A white background with text

Description automatically generated

A computer screen shot of a computer error

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a login page

Description automatically generated

### Setting Up Identity in Program.cs

In **Program.cs**, you can **configure Identity options** that control how users are registered and authenticated in your application. This includes **setting password strength**, **locking out users after too many failed attempts**, and **requiring email confirmation before allowing login**.

#### Program.cs Setup for Identity Options

You can set up Identity in your **Program.cs** by specifying options for password strength, lockout behavior, sign-in requirements, and more.

Here is how you can configure Identity in Program.cs:

A screenshot of a computer program

Description automatically generated

#### Explanation of Identity Options

**Password Settings:**

* We ensure that passwords are strong by requiring digits, uppercase and lowercase letters, and setting a minimum length of six characters

**Lockout Settings:**

* To protect against brute force attacks, users are locked out after five failed login attempts for five minutes

**User Settings:**

* We require unique emails for each user to avoid duplicate accounts

Sign-In Settings:

* For now, we do not require users to confirm their email before logging in, but this option can be enabled for better security

This configuration ensures that the application enforces good security practices while making user management straightforward.

### Watchlist

In this section, we will implement the **Watchlist** feature using a **mapping table** called **UsersMovies**. This table will represent the **many-to-many relationship between users and movies**, allowing **logged-in users to add movies to their watchlist** and manage their preferences.

#### Introducing the Watchlist Feature

The **Watchlist** will involve the following:

* A **UsersMovies** mapping table that links users and movies
* Controller actions to **add and remove movies** from the watchlist
* A **view to display** the user's **watchlist**
* A button on the movie details page to **add movies to the watchlist**
* A watchlist **link in the navbar for logged-in users**

#### Creating the UserMovie Model

To create the watchlist functionality, we will define a model called **UserMovie**, which **links users to the movies** they want to save for later. The **DbSet** for this model will be called **UsersMovies.**

This will establish a many-to-many relationship between users and movies:

* A **user** can save multiple movies to their watchlist
* A **movie** can appear in the watchlists of multiple users

We will use a composite key of **UserId** and **MovieId** to represent this relationship efficiently

A screenshot of a computer code

Description automatically generated

#### Update the DbContext for UsersMovies

**In the DbContext:**

* **UsersMovies** is the **DbSet** that will allow us to query and manage the relationship between users and movies
* We define a **composite key** using both **UserId** and **MovieId** to uniquely identify each watchlist entry
* We configure the relationships:
  + A user can have many movies in their watchlist
  + A movie can be added to the watchlists of multiple users

A screenshot of a computer program

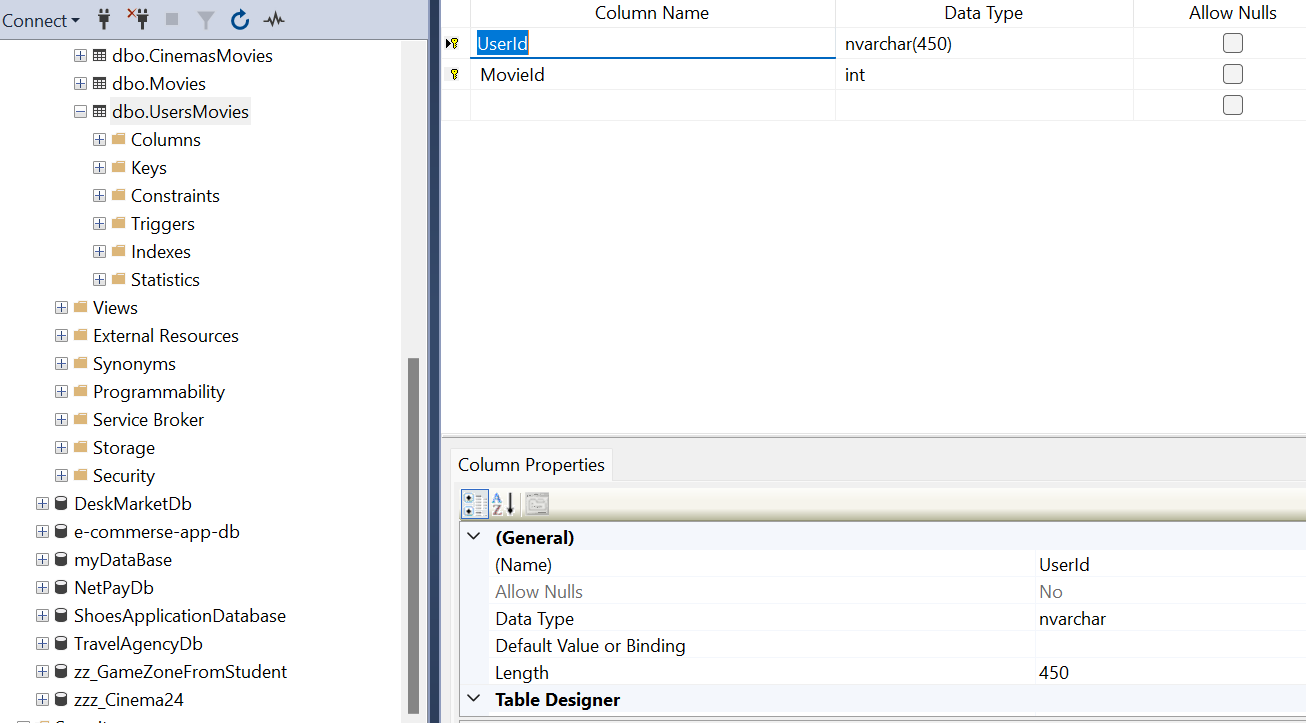
Description automatically generated

It is **NOT** strictly necessary to add collections in the Movie or IdentityUser models to represent the UserMovie relationship, because the **many-to-many relationship is already handled via the UserMovie mapping table** and **configured in the DbContext using the Fluent API.**

#### Add Migration and Update Database

**Create and apply the migration** for your updated **UserMovie** model and its relationship with the **Movie** and **IdentityUser** models.



#### Create WatchListViewModel

We need to create a new ViewModel that will represent the movies in the user’s watchlist. This ViewModel will hold only the essential data needed for displaying movies in the watchlist.

A screenshot of a computer program

Description automatically generated

This ViewModel helps us structure the data that we’ll use to display the movies in the user’s watchlist. Instead of passing the entire movie object, which contains more information than we need, we use the **WatchlistViewModel** to transfer only the required data between the controller and the view.

#### Extending the Movie Model

We’ve extended the **Movie** model by adding an **ImageUrl** property. This property will store the URL of the movie’s poster or image, which we can display in various parts of the application, like the watchlist.

A screenshot of a computer program

Description automatically generated

We use the **Add-Migration** command to **create a migration that reflects the changes** made to the **Movie** model.

The **Update-Database** command is used to **apply these changes to the database**, ensuring the **ImageUrl** column is created in the **Movies** table.

#### Update the MovieViewModel

A screenshot of a computer program

Description automatically generated

#### Update the Create Action in the MovieController

A screenshot of a computer program

Description automatically generated

#### Update the Create View

A screenshot of a computer program

Description automatically generated

#### Create the WatchlistController

The **WatchlistController** is responsible for handling all operations related to the user’s watchlist. It allows users to add movies to their watchlist, remove them, and view the current list of saved movies.

A screenshot of a computer program

Description automatically generated

**Actions:**

* **Index**: To display the user's watchlist
* **AddToWatchlist**: To add a movie to the user's watchlist
* **RemoveFromWatchlist**: To remove a movie from the user's watchlist

#### Creating the Watchlist Folder in Views

Now that we’ve set up the **WatchlistController** and the necessary actions, we need to create the corresponding views. First, we need to create a new folder called **Watchlist** in the **Views** directory, where we’ll store all the views related to the watchlist functionality.

Steps to Create the **Watchlist** Folder:

* In the **Solution Explorer**, right-click on the **Views** folder
* Select **Add** > **New Folder**
* Name the folder **Watchlist**

A screen shot of a computer

Description automatically generated

Once the folder is created, we can start adding views for actions like **Index**, which displays the movies in the user's watchlist.

#### Index Action

This action retrieves the logged-in user’s watchlist from the UsersMovies table:

* **It uses the WatchlistViewModel to pass only the necessary data (movie title, genre, and release date) to the view**

A screen shot of a computer program

Description automatically generated

#### Index View

Before moving to the next action, you can test the **Index** action by creating a view to display the movies in the user's watchlist.

A screen shot of a computer program

Description automatically generated



In this view, we are displaying the movie poster or image using the **ImageUrl** field. This makes the watchlist more visually appealing by showing an image alongside the movie's title, genre, and release date.

**Adding a Button for Logged-In Users to Add Movies to the Watchlist**

Now that the **Watchlist** is set up and accessible **via a route**, the next step is to allow users to add movies to their watchlist. However, this functionality **should only be available for logged-in users**.

To achieve this, we will add a button labeled **Add to Watchlist** next to each movie on the movie listing page. This button will only be visible to users who are logged in. When clicked, the button will send a request to the server to add the selected movie to the user's watchlist.

The **AddToWatchlist** action in the **WatchlistController** ensures that the movie is not already in the user's watchlist before adding it, preventing duplicate entries. This makes the application more dynamic by allowing users to manage their personal watchlists.

This functionality enhances the overall experience by providing personalization for logged-in users, making the app more engaging.

A screenshot of a computer

Description automatically generated

In the **Index** view for movies (e.g., **Views/Movie/Index.cshtml**), we'll add a button for **logged-in users** to   
**add movies to their watchlist**

**Updated View (All Movies Page):**

**A screenshot of a computer program

Description automatically generated**

****

* We added an **Add to Watchlist button** next to the movie details and program buttons
* The Add to Watchlist button is **only visible to logged-in users** by checking **User?.Identity?.IsAuthenticated**
* The button sends the **movieId** to the server, where it is handled by the **AddToWatchlist** action in the **WatchlistController**

A close-up of a computer screen

Description automatically generated

#### Add a Watchlist Link for Logged-in Users

To add a **Watchlist** link for logged-in users in the navigation bar, we can conditionally display the link based on the user's authentication status.

Here’s how we can update the navigation bar to include a **Watchlist** link for logged-in users:

A screenshot of a computer

Description automatically generated

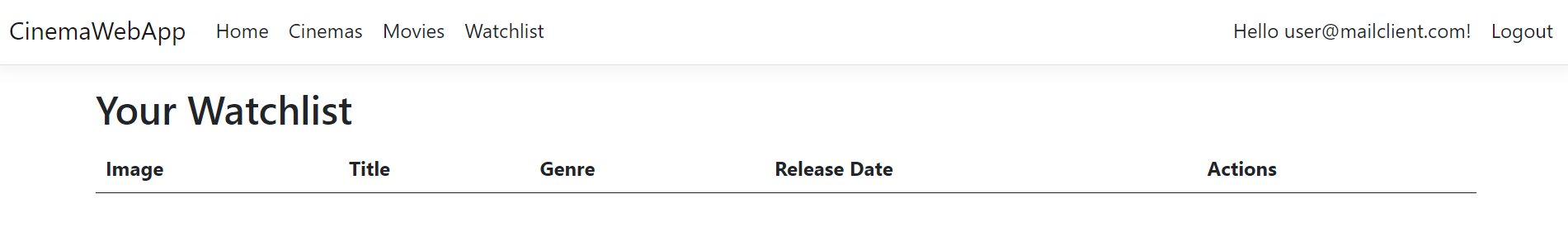


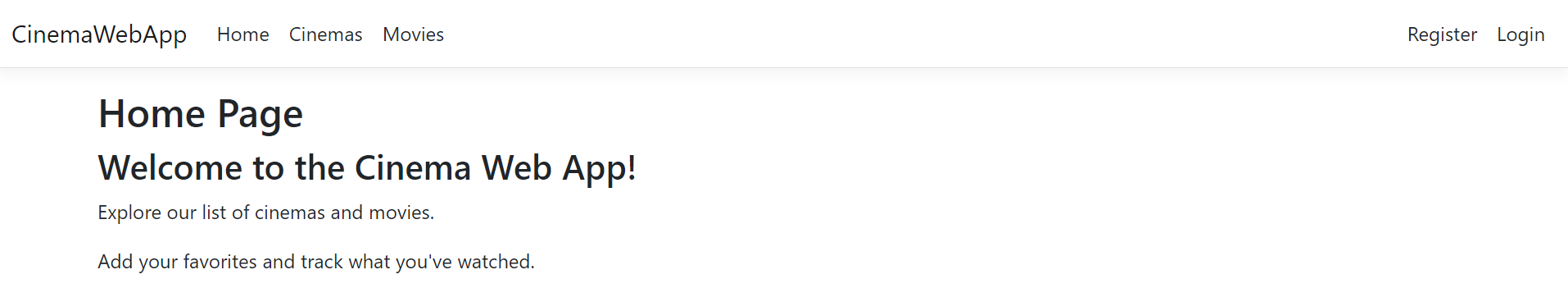
We’ve added a **Watchlist** link to the navigation bar, but it’s only displayed to logged-in users. This is done using the conditional check User?.Identity?.IsAuthenticated. If the user is authenticated, the **Watchlist** link will appear, allowing them to access their watchlist.

This ensures that only authenticated users can access their personalized watchlist while keeping the UI dynamic.

A white rectangular object with a black border

Description automatically generated





#### Adding a Movie to the Watchlist

To complete the logic for adding a movie to the watchlist, ensure that the **AddToWatchlist** action is in place in the **WatchlistController**. This action will save the movie for the currently logged-in user.

A screenshot of a computer program

Description automatically generated

Explaining the **AddToWatchlist** Action:

* **Getting the User ID**
  + Retrieves the unique ID of the currently logged-in user. This ID is used to associate the movie with the user in the watchlist.
* **Checking for Existing Entries** 
  + Before adding the movie to the watchlist, the action checks if it’s already present. This is done by querying the **UsersMovies** table to see if the combination of **userId** and **movieId** already exists
* **Adding the Movie to the Watchlist**
  + If the movie is not already in the watchlist, the action creates a new UserMovie object and adds it to the UsersMovies mapping table. This associates the movie with the logged-in user
* **Saving Changes and Redirecting**
  + After saving the changes to the database, the user should remain on the same page, so is redirected to **Movies/Index** page

A screen shot of a computer

Description automatically generated

#### Add a "Remove from Watchlist" Button in the Watchlist View

Now that users can add movies to their watchlist, we need to provide a **way for them to remove movies from the watchlist**. In this section, we will walk through how to implement this feature step by step.

**Add a "Remove from Watchlist" Button in the Watchlist View**

* The first step is to provide users with a button on their watchlist page to remove movies. The button will **send a POST request to our application**, which will **trigger the action to remove the movie**
* Open your Watchlist View
* **Add a form with a Remove button** for each movie in the watchlist

A computer code with text

Description automatically generated with medium confidence



#### Add the RemoveFromWatchlist Action in the Watchlist Controller

Next, we need to implement the **RemoveFromWatchlist action** in the **WatchlistController**. This action will handle the logic for **removing the selected movie from the watchlist:**

A computer code with many colorful text

Description automatically generated with medium confidence

Now users can manage their watchlist by both adding and removing movies as they please!