# Exercises: ASP.NET Core Introduction II

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 in an ASP.NET Core – Expanding the Application

In this exercise, we will continue **expanding our CinemaWebApp** by **adding more models** and implementing advanced features. As we move forward, we will introduce **the concept of View Models** and demonstrate how they can be used to better **manage the flow of data between the application layers**. View Models provide a clear **separation of concerns**, ensuring that only the necessary data is passed to and from the view.

Additionally, we will focus on **validating data** and **model validations** to ensure that user input is correct and secure. This will include both **server-side** and **client-side validation** using ASP.NET Core’s built-in features. By adhering to these **best practices**, we will enhance the **architecture of our application**, setting the foundation for **scalability** and **maintainability**. Throughout this workshop, we will also discuss **good practices in ASP.NET Core application** architecture, emphasizing clean code and a robust structure.

### What is a ViewModel?

A diagram of a model

Description automatically generated

In an ASP.NET Core MVC application, a **ViewModel** is a special type of model designed specifically to transfer data between the **View** and the **Controller**.  
It helps in separating the data you want to present on the UI from the actual **Domain Model** (such as the **Movie** or **Cinema** models, which represent your database entities). The ViewModel contains only the data that is necessary for a particular view, making it an essential part of the **Model-View-Controller (MVC)** architecture.

### MovieViewModel

Now that we understand what a **ViewModel** is and why it is important, let's create our first **ViewModel** for the **Movie** entity in our **CinemaWebApp**. This **MovieViewModel** will allow us to control what data gets passed between the view and the controller.

#### ViewModels Folder

* Right-click on the **Models** folder (or the root of your project if preferred)
* Select **Add** > **New Folder**
* Name the folder **ViewModels**

This folder will store all the ViewModels:

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

#### Add the MovieViewModel Class

* Right-click on the ViewModels folder and select **Add > Class**
* Name the class **MovieViewModel.cs**

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

#### Define the MovieViewModel

* In **MovieViewModel.cs**, define the properties that will be used in the view. Here's an example:

A screenshot of a computer program

Description automatically generated

This **ViewModel is now ready to handle data transfer** between the view and the controller.

#### BONUS: Set a Default Date in the ViewModel

* You can set a default date for the **ReleaseDate** in the **MovieViewModel**, such as the current date, to prevent the calendar from defaulting to year 0001:

A screenshot of a computer program

Description automatically generated

* Alternatively, you can set a default value directly in the **Create.cshtml** view using the value attribute:



#### Using the MovieViewModel

Now that we’ve created the **MovieViewModel**, we will demonstrate how to use this ViewModel to interact with the **Movie** entity in the application:

Modify the **Create (GET) Action - In the MovieController, the GET version of the Create action will pass an empty MovieViewModel to the view**

A close-up of words

Description automatically generated

Modify the **Create View** to Use **MovieViewModel**

* Since the **Create action** in the **MovieController** now uses the **MovieViewModel**, we need to update the **Create.cshtml** view to **bind to the MovieViewModel** and display the necessary form fields

A screenshot of a computer program

Description automatically generated

Update the **Create POST Action** in MovieController - In the **MovieController**, we will update the **POST** version of the Create action to work with the **MovieViewModel** and handle form submissions:

A screen shot of a computer program

Description automatically generated

**Explanation:**

* **viewModel**: The **MovieViewModel** is **passed as a parameter** from the form in the **Create.cshtml view**
* **Mapping the ViewModel to the Movie entity**: We manually map the properties from the **MovieViewModel** to the corresponding **Movie** entity
* **\_context.Movies.Add(movie)**: The new movie is added to the database
* **\_context.SaveChanges()**: Changes are saved to the database
* **RedirectToAction("Index")**: After successfully creating the movie, we redirect the user back to the movie list (**Index** action)

### Data Validation

Now that we have a **MovieViewModel** and a working **Create** form, it's essential to ensure that the data entered by users is valid before saving it to the database. We'll use **data annotations** for server-side validation and also enable **client-side validation** to improve user experience:

#### Adding Data Annotations for Validation

Update the MovieViewModel to **include validation rules for each property**. This will help us control what data is accepted in the form.

A screenshot of a computer screen

Description automatically generated

**Explanation of Data Annotations**:

* **[Required]**: Ensures that this field cannot be left blank. If it is, an error message will be shown
* **[StringLength]**: Restricts the length of the string field (e.g., movie name cannot be more than 100 characters)
* **[DataType(DataType.Date)]**: Ensures that the **ReleaseDate** field accepts valid dates only
* **[Range]**: Specifies a valid range for the **Duration** field (in this case, between 1 and 500 minutes)

#### Modify the Create (Post) Action to Handle Validation

In the **MovieController**, the **Create POST method** needs to **check if the form data is valid before proceeding to save** it:

A screenshot of a computer program

Description automatically generated

**Explanation:**

* **ModelState.IsValid**: This checks if the data provided by the user meets all the validation requirements specified in the **MovieViewModel**
* If the data is valid, it proceeds to create a new **Movie** entity and save it to the database
* If the data is invalid, it **returns the view with validation errors**, showing the user what needs to be fixed

#### Update the Create View to Display Validation Errors

To provide feedback to the user when input is invalid, we need to update the **Create.cshtml** view to display validation messages.

A screenshot of a computer program

Description automatically generated

Explanation:

* **asp-validation-for**: This tag helper displays validation messages for each form field if the validation fails
* **text-danger**: A Bootstrap class used to style the error messages in red

#### Test the Validation

* **Run your application**
* Navigate to **/Movie/Create**
* Test the form by
  + Submitting the form with **missing required fields**
  + Entering **invalid data** (e.g., a movie name that's too long or an invalid date)
* You should see validation **error messages appear** (after you submit the form with invalid data)

A screen shot of a movie

Description automatically generated

### Implementing Cinema in our CinemaWebApp

#### Create the Cinema Model

In your **Models** folder, add a new class called **Cinema.cs**

A screenshot of a computer

Description automatically generated

#### Create the CinemaMovie Join Entity

Since this is a **many-to-many** relationship, you need a join entity to connect the **Cinema** and **Movie** models. Create a new class called **CinemaMovie.cs**

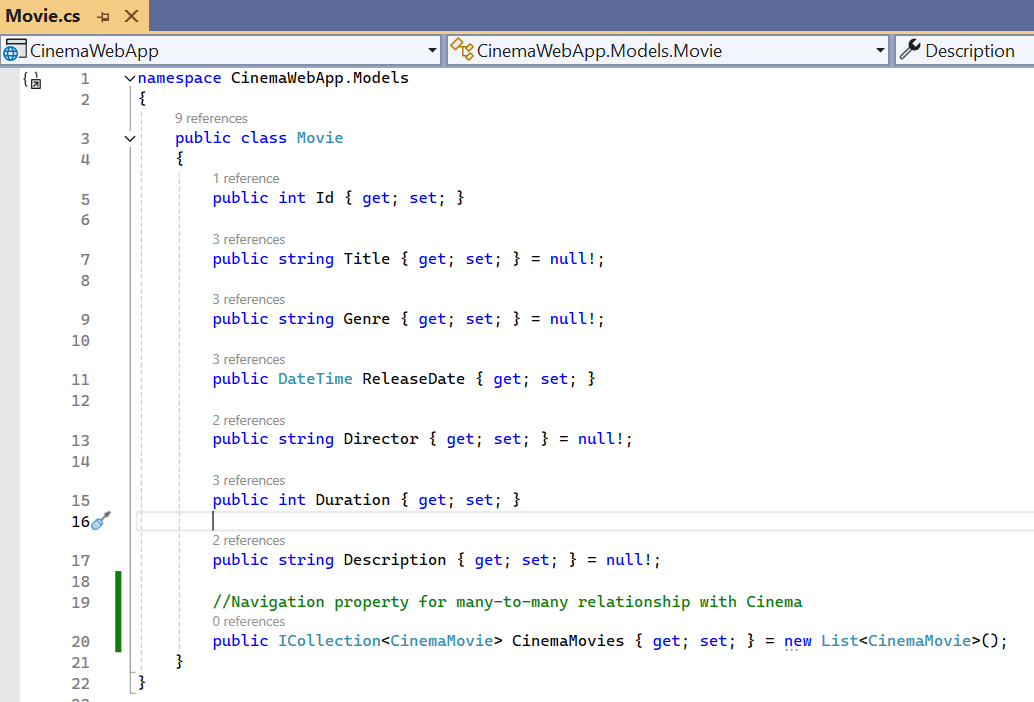
A screenshot of a computer program

Description automatically generated

This join entity will hold the **foreign keys for both Cinema and Movie**, establishing the **many-to-many relationship**

#### Update the Movie Model

You also need to **update the Movie model** to **reference the CinemaMovie join entity**. Open the **Movie.cs** file and modify it like this:



#### Update the AppDbContext

Now that we have both **Cinema**, **Movie**, and the **CinemaMovie** join entity, we need to configure the relationships in the **AppDbContext** class.

* Adding New **DbSets** for **Cinema** and **CinemaMovie**

In Entity Framework, each entity (class) that you want to interact with in the database needs to be represented by a **DbSet** in your **AppDbContext**

* + **DbSet<Movie> Movies**: This is the table that holds all **Movie** records
  + **DbSet<Cinema> Cinemas**: This is the table that holds all **Cinema** records
  + **DbSet<CinemaMovie> CinemasMovies**: This is the **join table** for the many-to-many relationship between **Cinema** and **Movie**

A group of text on a white background

Description automatically generated

* Configuring the **Many-to-Many Relationship** Using **Fluent API**

By default, **Entity Framework Core can infer some relationships using conventions**, but in more complex scenarios (like a **many-to-many relationship**), we need to **provide more explicit instructions**. **Fluent API** is a way to **configure relationships and behavior in your model classes through code**, without using attributes in your models

What are we configuring using Fluent API?

* **CinemaMovie as a join table**: We define **CinemaMovie** as the **join table** for the many-to-many relationship between **Cinema** and **Movie**
* **Composite primary key**: We configure **CinemaMovie** to use a composite primary key (a key composed of both **CinemaId** and **MovieId**)
* **Relationships**: We configure each side of the relationship, telling Entity Framework how **Cinema**, **Movie**, and **CinemaMovie** relate to each other

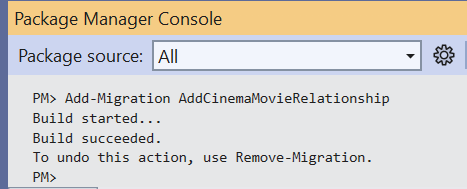
Here’s how the **OnModelCreating** method works:

A screen shot of a computer program

Description automatically generated

#### Creating and Applying the Migration

Now that **we've updated the AppDbContext** with the **Cinema**, **Movie**, and **CinemaMovie** entities, it's time to **create and apply a new migration to update the database schema**. This will ensure that the database reflects the changes we’ve made in our models and relationships:



**This command will**:

* Create a new migration file that includes the SQL required to add the **Cinema**, **Movie**, and **CinemaMovie** tables, and establish the relationships between them
* The migration file will be created in the **Migrations** folder

**Run the Update-Database Command:**

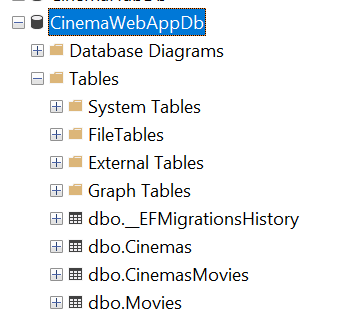
* In the Package Manager Console, type the following command and press **Enter**

A computer screen shot

Description automatically generated

**This command will**:

* Apply the migration to the database
* Create the new tables (**Cinemas**, **Movies**, and **CinemasMovies**) and establish the many-to-many relationship between **Cinema** and **Movie**



**Verify the Database**

* The **CinemasMovies** table should have **two foreign keys**: one to the **Cinemas table (via CinemaId)** and one to the **Movies table (via MovieId)**



#### Creating the CinemaController

Let's start by creating an empty **CinemaController** and implement each action step by step.

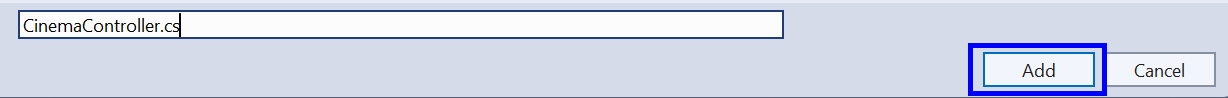
**Create an Empty CinemaController**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer error

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated**

**Explanation:**

* The **Index action fetches all the cinemas** **from the database** using the **AppDbContext** and returns them to the **Index view**

#### Creating the CinemaIndexViewModel

Here’s an example of a simple **CinemaIndexViewModel**. It might **include only properties that are needed** in the views, such as the cinema’s name and location

A screenshot of a computer

Description automatically generated

#### When Do We Need Id in the ViewModel

**For Edit, Delete, or Details Actions:**

* If you're going to allow users to **edit**, **delete**, or **view detailed information** about a cinema, you need the **Id** to identify the specific cinema. The **Id** serves as the unique identifier for routing and database operations

**Example:**

* If you create a link to the **Edit** page for a cinema, the URL would include the cinema's **Id**, like this: **/Cinema/Edit/1**

**Updated Index Action to Use CinemaIndexViewModel**

Here’s how we can refactor the **Index** action in the **CinemaController:**

A screen shot of a computer code

Description automatically generated

#### Create the Index View

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated



#### Detailed Explanation of the View

**Model Declaration:**

* The view expects a model of type IEnumerable<CinemaIndexViewModel>, which is a list of cinema objects. The **CinemaIndexViewModel** contains the cinema's **Id**, **Name**, and **Location**, which are used to populate the content of the view



**Heading for the Cinemas Section:**

* A simple heading to label the section displaying the list of cinemas



**Button to Add a New Cinema:**

* This button links to the **Create** action in the **CinemaController**, which allows users to add a new cinema
* The **asp-action="Create"** tag helper generates the appropriate link to the **Create** action
* The button has Bootstrap classes:
  + **btn-success**: A green button style
  + **mb-4**: Adds a margin below the button to create space between the button and the list of cinemas



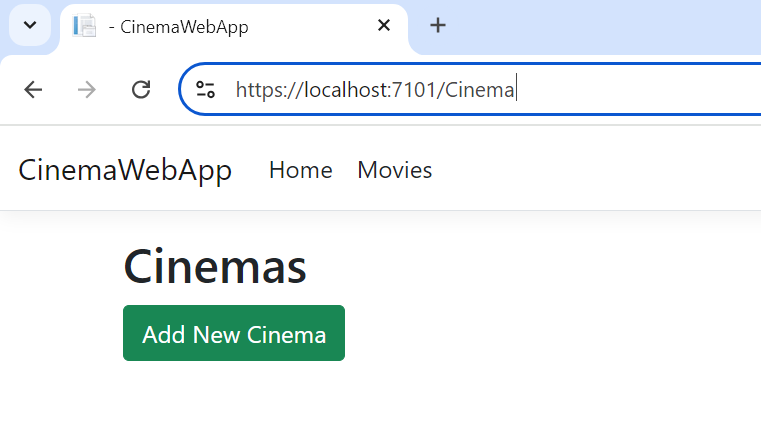
**Cinema Cards in a Bootstrap Grid:**

* Bootstrap Grid Layout:
  + The outer **row** class creates a grid container that holds the cinema cards
  + **col-md-4**: This class defines that there will be three columns per row on medium screens and larger (i.e., for desktop view)
* Each Cinema as a Card – **foreach** loop:
  + The loop goes through each **cinema** in the list of **CinemaIndexViewModel** objects passed from the controller
  + For each cinema, a Bootstrap **card** is created
  + Give the image an easy-to-remember name, like **my-cinema.jpg**
  + In the **Index.cshtml** file, replace the default image path (cinema-default.jpg) with the path to your new image

**Adding Your Own Cinema Image:**

* By default, we’re using a placeholder image for each cinema (**cinema-default.jpg**). However, you can download and use your own images for the cinemas to personalize the look of the application
* Download an image that you’d like to use for your cinemas. Make sure it is an appropriate size (e.g., 400x300 pixels) to fit well in the card layout
* Once you’ve downloaded the image, save it in the **wwwroot/images** directory of your project
* If the **images** folder doesn’t exist, you can create it

A screenshot of a computer

Description automatically generated

#### Create the Create Action in the CinemaController

Add the **Create** action that will handle both **GET** (to display the form) and **POST** (to handle form submissions)

In your **CinemaController**, add the following methods:

* **GET** Action

A close-up of text

Description automatically generated

* **POST** Action

A screen shot of a computer code

Description automatically generated

Explanation:

* **GET Action**:
  + Renders the **Create.cshtml** form for adding a new cinema
* **POST Action:**
  + Accepts the **CinemaViewModel** submitted from the form
  + If the form data is valid (passes model validation), it creates a new Cinema object and adds it to the database
  + If the data is invalid, it re-renders the form with validation error messages

#### CinemaCreateViewModel

A screenshot of a computer program

Description automatically generated

#### Create the Create.cshtml View

Add the Following Code to the **Create.cshtml** View:

A screenshot of a computer screen

Description automatically generated



#### Test the Add Cinema functionality

Now that the view is ready, you can run the application to test the **Add Cinema** functionality and ensure that the view looks correct and works as expected:

A screenshot of a computer

Description automatically generatedA screen shot of a movie camera

Description automatically generated

#### Add Cinemas to the Navbar

**In Solution Explorer:**

* Navigate to **Views/Shared**
* Open the **\_Layout.cshtml** file

**Find the Navbar Section:**

* Look for the navigation items inside the **<nav>** tag, typically under a <ul> element with navbar-nav class. You'll see items like **Home** and **Movies**
* Add a new <li> element to represent the **Cinemas** section in the navigation bar. Here's how it should look:

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### Cinema Movie Program

This section will **allow users to view the movie program** for a specific cinema.

#### Create the MovieProgramViewModel

This view model will represent the **movies that are part of the cinema's program**, focusing on the **Title** and **Duration** of the movies.

A screenshot of a computer program

Description automatically generated

#### Create the CinemaDetailsViewModel

**CinemaDetailsViewModel** will represent the cinema's details along with **the list of movies associated with that cinema.**

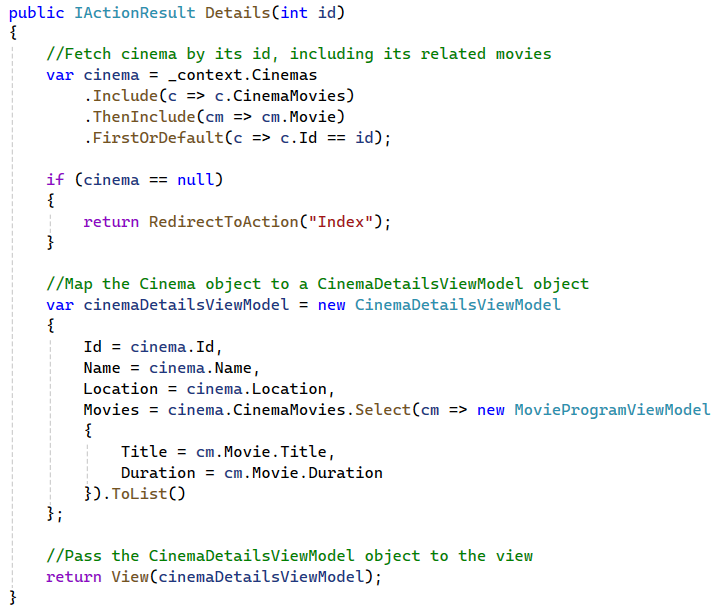
**A computer code with text

Description automatically generated with medium confidence**

#### Create a Details Action in CinemaController

**Add the Details Action in the CinemaController:**

* Here’s how the **Details** action might look, using both the **CinemaDetailsViewModel** and the **MovieProgramViewModel**



#### Cinema Details View

**Ensure the Details.cshtml View is Ready:**

* Make sure that the Details.cshtml view is set up to display the movie program using the CinemaDetailsViewModel. If you’ve followed the earlier steps, it should look like this:

A screenshot of a computer screen

Description automatically generated



A screenshot of a computer

Description automatically generated

#### Cinema Movies Program: No Movies Added Yet

As you can see, when you navigate to the **Cinema Movie Program** page, there are no movies currently showing for any cinema. This is why the message "No movies are currently showing at this cinema" is displayed.

This happens because we haven't yet linked any movies to the cinemas in our application. In the next steps, we will start implementing the logic to add movies to specific cinemas, allowing you to manage and display the movie programs for each cinema.

### Add Movie to Cinema Program

To move forward, we will implement the following logic:

* **Link Movies to Cinemas**: We’ll create functionality to assign movies to specific cinemas
* **Update the Movie Program**: Once we add movies to cinemas, the **Movie Program** section will display the list of movies currently showing at the selected cinema
* We will implement the **"Add to Program"** button in the **All Movies** section

A screenshot of a computer

Description automatically generated

#### Update All Movies View

In the **All Movies** view, we'll add the **"Add to Program"** button for each movie. Let’s assume the view is called **Index.cshtml** in the **Movies** folder

A screenshot of a computer

Description automatically generated

**Add the Button to the Actions Section**:

A computer screen shot of a computer code

Description automatically generated

Explanation of Changes:

* A new button is added next to the **Details** button for each movie in the actions column
* The **"Add to Program"** button links to the **AddToProgram** action in the **MovieController**, passing the **movieId** as a route parameter using **asp-route-movieId="@movie.Id"**

A screenshot of a computer

Description automatically generated

#### CinemaCheckBoxItem View Model

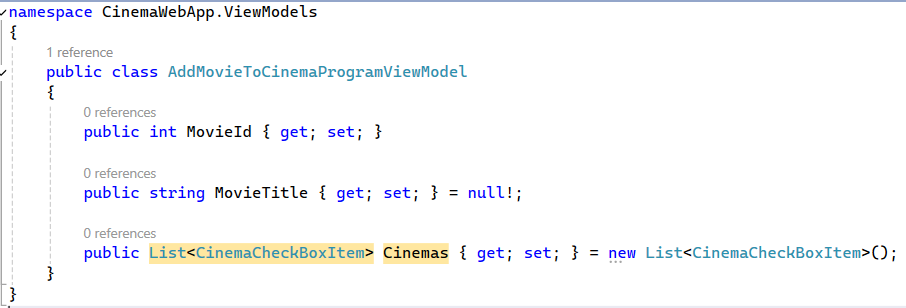
The **CinemaCheckBoxItem** view model is a simple class that represents a cinema as a selectable item (using checkboxes) when assigning a movie to one or more cinema programs. This view model is used to manage which cinemas are selected by the user.

A screenshot of a computer

Description automatically generated

#### AddMovieToCinemaProgramViewModel

This view model is responsible for carrying all the data required to display the movie and the list of cinemas where users can assign the movie to their program. It also facilitates passing the user's selections back to the controller when the form is submitted.



#### Add Movie Action in MovieController

Add the **AddToProgram** (GET) Action in the **MovieController:**

* Plan:
  + **We pass the movie ID** when the user clicks **Add to Program**
  + **The view will show the movie title** and allow the user to select cinemas
* Since the movie is already known when clicking the **Add to Program** button, we don't need a dropdown to select a movie. Instead, we'll pass the movie ID directly from the **AddToProgram** button and display the movie title on the **AddToProgram** view.

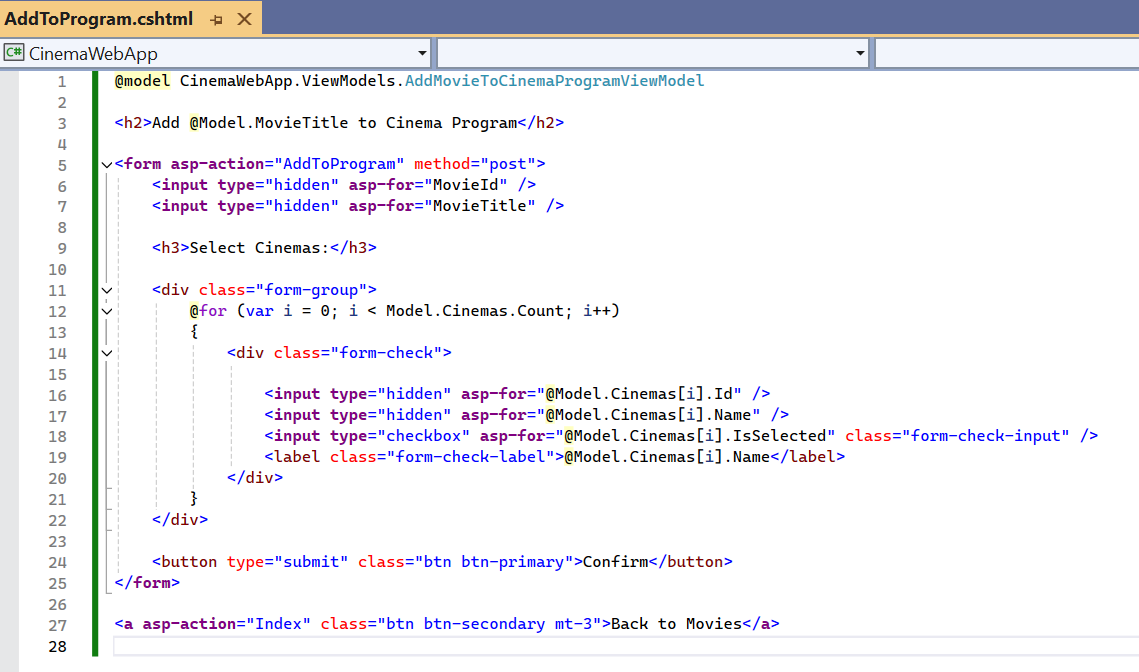
A screenshot of a computer program

Description automatically generated

**How the Data is Used in the View:**

* When this data reaches the view (**AddToProgram.cshtml**), it’s used to display the movie title and the list of cinemas with checkboxes





A screenshot of a computer

Description automatically generated

Add the **AddToProgram** Action (POST) in the **MovieController**

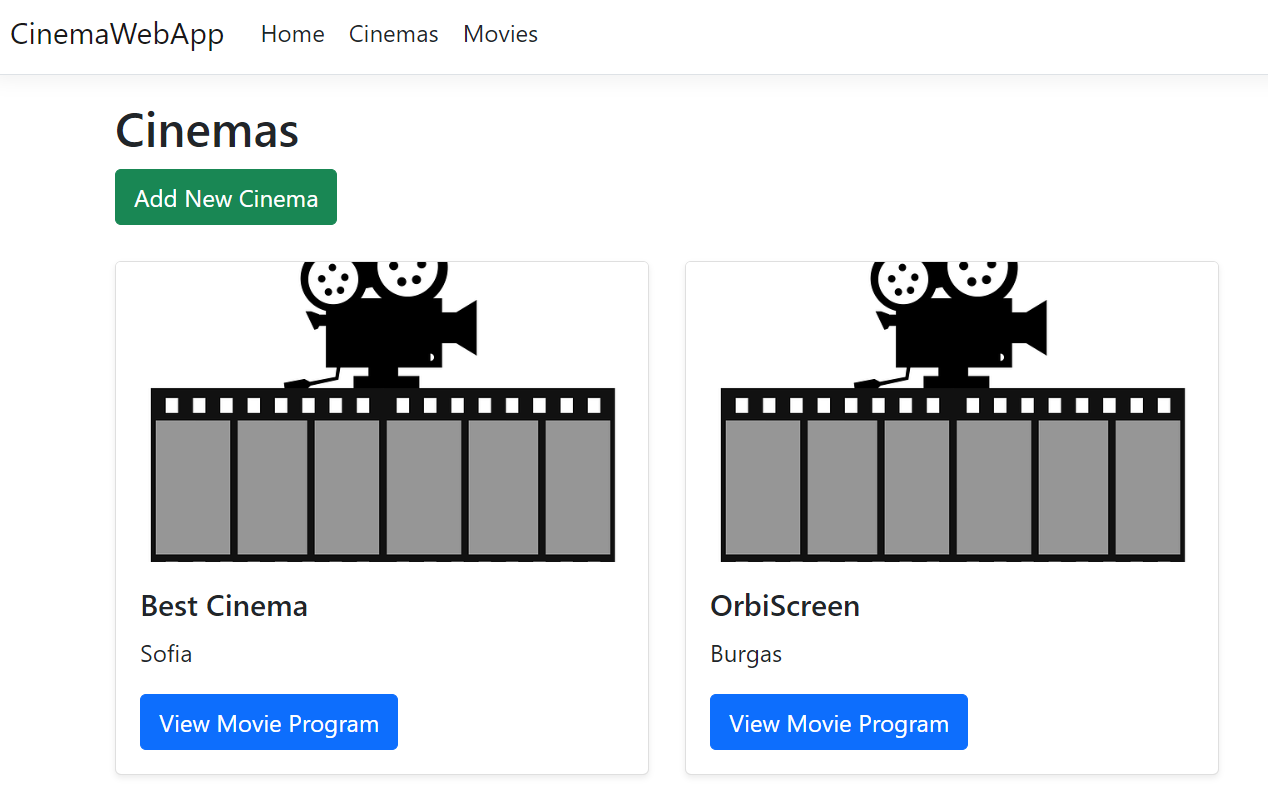
* Let's implement the **POST** method for the **AddToProgram** action. This method will process the form submission, take the selected cinemas, and assign the movie to those cinemas
* This method will:
  + Validate the input data
  + Remove any existing movie-to-cinema assignments for this movie
  + Add new assignments based on the selected cinemas
  + Save the changes to the database and redirect the user

A screen shot of a computer program

Description automatically generated

**Add Movies and Cinemas**

* Ensure you have a few movies and cinemas in your database so you can test the **"Add to Program"** functionality.
* Check if you can add a movie and associate it with one or more cinemas using the form.



A screenshot of a computer

Description automatically generated

