# Exercises: ASP.NET Core Introduction

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

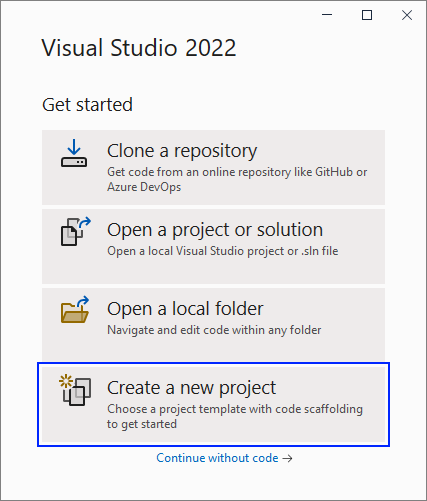
A movie ticket and popcorn

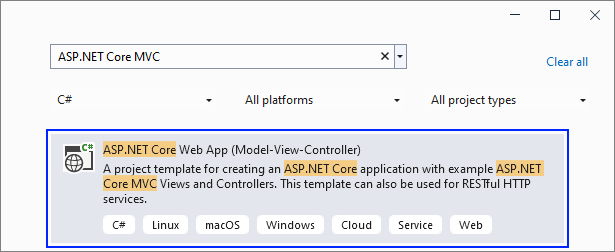
Description automatically generated

## Create CinemaWebApp in an ASP.NET Core

Open **Visual Studio 2022**. You will see the start screen with various options to get started.

Select [Create a new project] from the list of options. This will initiate the process of creating a new project from scratch.  
On the next step, **choose** [ASP.NET Core Web App (Model-View-Controller)] as a **project template**. The steps are shown below:



****

Give a **name** to your project and solution:

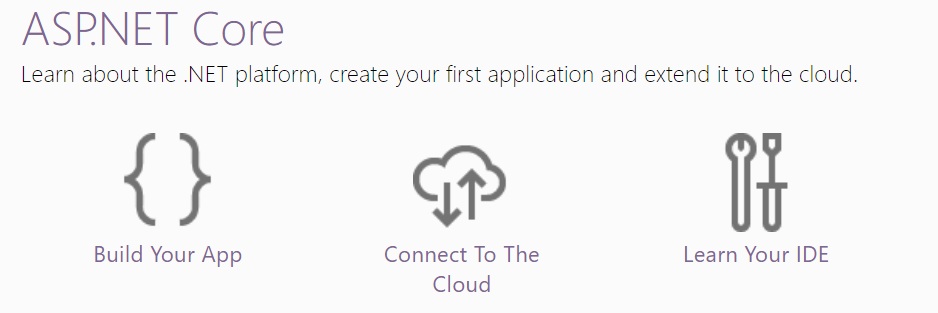
A screenshot of a computer

Description automatically generated

On the next step you should **choose** your target frameworok and click on the [Create] **button**:

**A screenshot of a computer

Description automatically generated**



Now your **app is created** and looks as shown below. Note that it has **folders** for **controllers**, **models** and **views** because of the template we chose:

A screenshot of a computer

Description automatically generated

If you **run the app**, you will see the **default** "Home" **page**, which is served by the HomeController in the app:

A screenshot of a web browser

Description automatically generated

### What is a Controller?

A **Controller** is a key part of the **MVC pattern**. It is responsible for handling incoming requests, interacting with the **Model** to retrieve or manipulate data, and then returning the appropriate **View** to display the data to the user.

### HomeController Pages

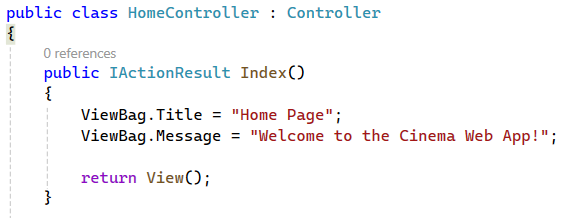
#### Modify the "Home" Page

Now we want to **modify** the "Home" **page** to look like this:

A screenshot of a web page

Description automatically generated

Change the Index() **method** of the HomeController to **change the page**. The **controller action** should return a **view**, as it does already, but also use the ViewBag **class** to **create a message**, which will be **used in the view**. Modify the method like this:



Now you should modify the Index.cshtml **view** in the "/Views/Home" **folder** to **display the page differently**. Use the ViewBag **class** to **get the message** from the controller. Note how the Razor **views** allow us to use **C# code** inside **HTML**:

A screenshot of a computer

Description automatically generated

**Run the app** with [Ctrl] + [F5] and make sure the "Home" **page** looks as shown on the screenshot above.

#### Simplify the HomeController

**Remove the Privacy action**, you don’t need a separate Privacy page right now.   
**Remove the Error action** unless you want custom error handling (this can be added later).  
**Clean up** any unnecessary using statements.

Here’s the simplified version of your HomeController:

A screenshot of a computer

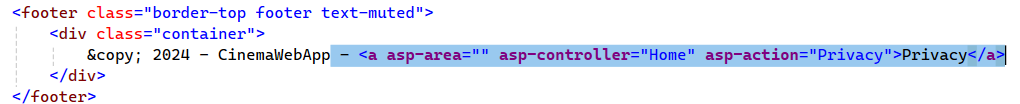
Description automatically generated

#### Simplify the Navbar

To simplify the navbar and remove redundant links, like the **Privacy** link that you no longer need, we can clean up the code:

A computer screen shot of a computer code

Description automatically generated



* **Removed the Privacy link** from both the navbar and the footer
* Left only the **Home** link in the navbar, which points to the Home page

### Creating Our First Model

#### Movie Model

**Navigate to the Models Folder**: In **Solution Explorer** (the panel on the right), find the folder named Models.

* Right-click the **Models** folder, select **Add** > **Class**
* A window will pop up asking for the name of the class. Name it **Movie.cs** and click **Add**

A screenshot of a computer

Description automatically generated

Once the class is created, you will see a blank class file. This is where we define the structure of our **Movie** model

#### Define the Movie Model

**Explanation of the Properties:**

* **Id** - A unique identifier for each movie. This will be automatically managed by the database
* **Title** - This is a string property that stores the name of the movie (e.g., "The Matrix")
* **Genre** - A string property that stores the genre (e.g., "Action", "Comedy", "Drama")
* **ReleaseDate** - This will store a DateTime object, but only the **month** and **year** will be relevant. The **day** can be ignored or set to a default value (such as the first day of the month)
* **Director** - The name of the director who directed the movie (e.g., "Christopher Nolan")
* **Duration** - The length of the movie in minutes (e.g., 120 for a two-hour movie)
* **Description** - A short summary or synopsis of the movie

A screenshot of a computer

Description automatically generated

We have successfully created the **Movie** model, which contains all the essential information about a movie. This model will serve as the **foundation for storing and retrieving movie data** in our application.

### Create a Model Controller

The next step will be to create a **MovieController**. This **controller will handle user interactions** with the movie data, such as adding new movies, viewing a list of movies, and managing movie details. It will act as the **bridge** between our **Movie** model and the views that display movie information to the users.

For the Movie entity, we need a controller that allows us to:

* List All Movies
* Add New Movie
* View Details of a Single Movie

#### Navigate to the Controllers Folder

In **Solution Explorer**, right-click the **Controllers** folder. Select **Add** > **Controller**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A white rectangular object with blue border

Description automatically generated

In **Solution Explorer**, you should see your newly created **MovieController** under the **Controllers** folder:

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

#### Add Basic Controller Structure

Here's the basic structure of the controller, which will handle requests related to movies:

* **Include** all needed libraries and relationships A screenshot of a computer

  Description automatically generated
* Create an **in-memory** sequence to store movies A screen shot of a computer code

  Description automatically generated
* **Index Action**: This method retrieves all movies and sends them to the **Index** view. The list of movies will be displayed on the homepage for movies A computer screen shot of a computer program

  Description automatically generated
* **Create (GET)**: The Create method (with **[HttpGet]**) displays the form for adding a new movie A screenshot of a computer

  Description automatically generated
* **Create (POST)**: This **[HttpPost]** method handles the form submission when creating a new movie.   
  It **assigns** an **ID to the new movie**, adds it to the list, and redirects back to the **Index** page A screen shot of a computer program

  Description automatically generated
* **Details Action**: This action takes an **id parameter**, searches for the movie with that id, and displays its details. If no movie is found, it returns a **NotFound()** result. A screen shot of a computer code

  Description automatically generated

### Create Views for the Specific ModelController – Movie Controller

#### Creating the "Create" View (GET and POST)

The **GET** method is used to display the form for adding a new movie. In the **MovieController**, you already have the following method:

A close-up of a computer code

Description automatically generated

This **action** simply returns the **Create** view, which will display the **form for entering movie details**.

**In Solution Explorer**:

* Navigate to **Views > Movies** (create a **Movies** folder inside **Views** if it doesn't exist)
* Right-click the **Movies** folder, and select **Add** > **Razor View**
* Name the view **Create.cshtml**

A screenshot of a computer

Description automatically generated A screenshot of a computer screen

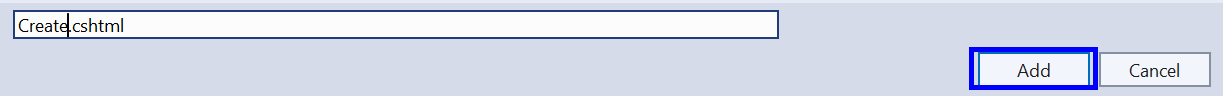
Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

**Create the Movie Form**: Replace the content of **Create.cshtml** with the following form:



A screenshot of a computer program

Description automatically generated

**Explanation:**

* The form uses the **asp-for** tag helpers to **bind the input fields** to the properties of the **Movie** model
* The **asp-action="Create"** specifies that the form will submit to the Create action in the **MovieController** (the POST version)
* After the form is submitted, the data will be processed by the **POST** method in the **MovieController**

**Run and Test the Create View:**

* Run your application
* Navigate to **/Movie/Create** to see the form and add a new movie
* After submitting, it should redirect you to a placeholder **Index** page (which we'll create next)

A screenshot of a computer

Description automatically generated

#### Creating the "Index" View (All Movies)

**Index Action in the MovieController**:

* First, let’s make sure the Index action is defined in the **MovieController**. This action will retrieve the list of all movies and pass it to the view: A screen shot of a computer

  Description automatically generated
* The **Index action** **passes the movies list** (which we’ve been using to simulate a database) to the view

**Create the Index View:**

**A screenshot of a computer

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated**

**List Movies in the Index View:** Replace the content of **Index.cshtml** with the following code:

****

A screen shot of a computer

Description automatically generated

**Explanation:**

* The model passed to the view is an **IEnumerable<Movie>**, which represents a list of movies
* The **@foreach loop** iterates through the list of movies and displays each one in a table row
* Each row contains movie details (name, genre, release date, and duration) and a link to view more details about the movie
* The **Add New Movie** button at the bottom links to the **Create** view

**Run and Test the Index View:**

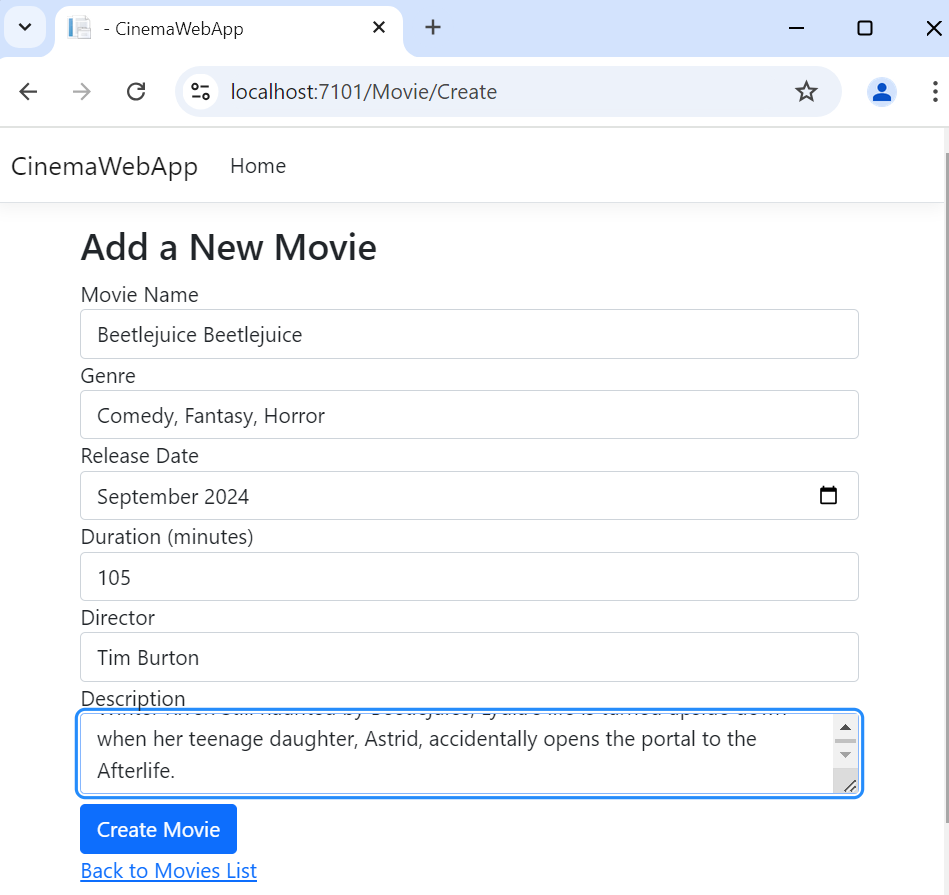
* Run your application
* Navigate to **/Movie/Index** or simply **/Movie** to see the list of movies you’ve added so far
* You should see the list of movies, with a **Details** button for each movie

**No Movies Yet?**

* Click the **"Add New Movie"** button on the All Movies page
* Fill in the details for the new movie in the **Create** form (e.g., title, genre, release date, director)
* Submit the form, and the movie will be added to your list

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

#### Creating the "Details" View

The **Details** view will allow users to view more information about a specific movie when they click the **Details** button from the movie list.

**Details Action in the MovieController**:

* First, let’s ensure the **Details** action is set up in the **MovieController**. This action will retrieve the movie by its id and pass it to the view  
  A screen shot of a computer program

  Description automatically generated
* This action retrieves the movie with the specified id from the list. If the movie is not found, it returns a **404 Not Found** error
* If the movie is found, it is passed to the **Details** view for display

**Create the Details View**:

A screenshot of a computer error

Description automatically generated

A screenshot of a computer

Description automatically generated

**Display Movie Details in the View**: Replace the content of **Details.cshtml** with the following code:



A screenshot of a computer program

Description automatically generated

**Explanation**:

* The view is strongly typed to the Movie model
* It displays various details about the movie, including its genre, release date (formatted to show only the month and year), duration, director, and description
* A **Back to All Movies** button is included to take the user back to the **Index** page

**Run and Test the Details View**:

* Run your application
* Navigate to the **All Movies** page and click the **Details** button next to any movie
* You should be taken to a page displaying all the details of the selected movie

A close-up of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

#### Pay Attention to the URL

When you click the **Details** button for a movie on the **All Movies** page, notice how the URL changes. For example, if you click on a movie with an ID of 1, the URL might look like this:

A screen shot of a computer

Description automatically generated

The number at the end of the URL (1 in this case) is the **ID** of the movie. This ID is passed to the **Details** action in the **MovieController**. The controller then uses this ID to find the specific movie in the list and pass its details to the view.

Each movie has a **unique** **ID**, and this ID is what allows the application to know which movie’s details to display. This pattern of passing an ID through the URL is common in MVC applications, and you'll see it frequently when working with data.

### Adding Links to the Navbar

Now that we have the functionality for movies in place, let’s add a link to the **Movies** section in the **navigation bar**, so users can easily access the list of all movies from any page.

#### Open the Layout File

In an ASP.NET Core MVC application, the **Layout file is responsible for defining the common structure** of your website, including the navigation bar.

In **Solution Explorer**, navigate to:

* **Views** > **Shared** > **\_Layout.cshtml**
* Open this file

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

#### Add the Movies Link to the Navbar

Find the existing navbar code inside the **<nav> tag**:

A computer code with blue lines

Description automatically generated with medium confidence

To **add Movies link**, **modify the code** as follows:

A screenshot of a computer code

Description automatically generated

**Explanation**:

* The **<a>** tag with **asp-controller="Movie"** and **asp-action="Index"** creates a link to the **Index** action in the **MovieController**, which lists all movies
* The link text is **"Movies"** which will be displayed in the navigation bar

**Run and Test**:

* Run your application
* You should now see a **Movies** link in the navbar
* Clicking the **Movies** link will take you to the page that lists all the movies (the **Index** view of the **MovieController**)

A screenshot of a computer

Description automatically generated

The **in-memory list** is temporary and **will reset every time the application restarts**. To persist the movies and connect the application to a real database, we need to set up a **Database** using **Entity Framework Core** (EF Core) and configure it to store movie data

### Connecting the Application to an External Database Server

#### Install Entity Framework Core Packages

**Open the NuGet Package Manager**:

* In **Visual Studio**, right-click on your project in **Solution Explorer** and select **Manage NuGet Packages**

**Installing the Required Packages**:

* **Microsoft.EntityFrameworkCore.SqlServer** (for SQL Server support)
* **Microsoft.EntityFrameworkCore.Tools** (for migrations)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

#### Set Up the Database Context

The **DbContext** class is responsible for interacting with the database.

**Create a New Context Class**:

* In the **Models** folder (or create a new **Data** folder), right-click and select **Add** > **Class**
* Name it **AppDbContext.cs**
* **Add a DbSet<Movies> collection**

A screenshot of a computer

Description automatically generated

#### DbContextOptions in AppDbContext

**What is DbContextOptions?**

* **DbContextOptions** is a class that carries configuration settings for the DbContext. It tells Entity Framework how to connect to the database, which provider to use (e.g., SQL Server), and any additional configuration for the database
* The options parameter contains this information when the **AppDbContext** is instantiated by ASP.NET Core. This is why we pass options to the base constructor (: base(options))

**Why is this needed?**

* **Separation of concerns**: By **injecting** the **DbContextOptions** into AppDbContext, you **decouple the database configuration from the database access logic**. This allows the configuration (such as the connection string) to be handled in **Program.cs** or another external file like **appsettings.json**

#### Configuring DbContext in Program.cs

In **Program.cs**, we add the following code:

A computer code with text

Description automatically generated with medium confidence

**What does this code do?**

**builder.Services.AddDbContext<AppDbContext>(...)**

* This registers the AppDbContext with the **dependency injection** (DI) system in ASP.NET Core
* It makes the **AppDbContext available as a service** so it **can be injected into controllers**, services, or other parts of the application
* Every time the application needs to interact with the database, it will **request the AppDbContext from DI**, and **ASP.NET Core will automatically provide a configured instance**

**options.UseSqlServer(...)**

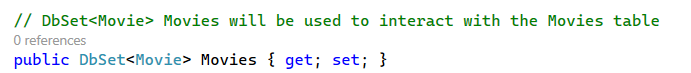
* This configures **Entity Framework Core** to use **SQL Server** as the database provider
* **builder.Configuration.GetConnectionString("DefaultConnection")** retrieves the connection string named "DefaultConnection" from **appsettings.json**, which contains the details about the SQL Server instance and database to use

**Why is this needed?**

* **DbContext and Database Provider**: Every DbContext needs to know which database provider to use (e.g., SQL Server, SQLite, etc.) and how to connect to the database. This configuration tells Entity Framework that we are using SQL Server and gives it the connection details to interact with the database

#### Understanding DbSet

In the **AppDbContext**, we see the following property:



**What is DbSet?**

* A **DbSet** represents a collection of all entities in the database of a specific type, in this case, Movie
* Think of a **DbSet** as a table in the database. The Movies **DbSet** represents the table where all movie records will be stored
* You can use the **DbSet** to query, add, update, or delete records from the corresponding table

**Why is this needed?**

* **DbSet** provides the mechanism to interact with the data stored in the database. When you perform operations like retrieving movies (\_context.Movies.ToList()), adding a new movie (\_context.Movies.Add(movie)), or deleting a movie, you are interacting with the **DbSet**
* **DbSet<Movie> Movies** tells Entity Framework that you want a collection of Movie entities that will be mapped to the database

#### Configuring *appsettings.json* for the Database Connection

In order to connect your application to a **SQL Server database** using **SQL Server Management Studio (SSMS)**, we need to configure the connection string correctly in the **appsettings.json** file. This will tell Entity Framework Core how to connect to the SQL Server instance and which database to use.

**Open** **appsettings.json:**

* In **Solution Explorer**, navigate to the **appsettings.json** file
* This file contains **key-value pairs for application configuration**, including the database connection string

**Add the Connection String:**

* You'll need to replace the default connection string (or add a new one if it's not there) to point to your **SQL Server** instance and database

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* You can see **different** Connection Strings **approaches** [here](https://www.connectionstrings.com/sql-server/):

A screenshot of a computer

Description automatically generated

**Creating the Initial Migration and Updating the Database**

Now that you have configured the connection string in **appsettings.json** and set up the **DbContext**, it's time to create the **initial migration and update the database**. This will create the necessary database tables based on your models (like **Movie**) and allow your application to interact with the database.

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

A screenshot of a computer

Description automatically generated

### Implementing the MovieController to Work with the Database

We will modify the **MovieController** so that it interacts with the **AppDbContext** instead of using an in-memory list. The **AppDbContext** will allow the controller to retrieve, add, and manage movies directly from the database.

#### Injecting the AppDbContext into the Controller

In **MovieController**, we need to inject the **AppDbContext** so that the controller can interact with the database.

**DbContext Injection**:

* The AppDbContext is injected into the **MovieController** through constructor dependency injection. This allows the controller to access the database  
  A screen shot of a computer code

  Description automatically generated

**Index Action**:

* The **Index()** method now **retrieves all movies** from the database using **\_context.Movies.ToList()** and passes them to the view. This **replaces the previous in-memory list**.  
  A close-up of a list

  Description automatically generated

**Create (GET and POST)**:

* The **Create() (GET) method** displays the form for adding a new movie
* The **Create(Movie movie) (POST) method** handles the form submission. If the model is valid, it adds the new movie to the database using **\_context.Movies.Add(movie)** and saves the changes with **\_context.SaveChanges()**

A screen shot of a computer program

Description automatically generated

**Details Action:**

* The **Details(int id) method** uses **\_context.Movies.Find(id)** to find a movie by its unique ID in the database. If the movie is not found, it returns a **404 Not Found** page

A computer screen shot of a program

Description automatically generated

#### Run and Test the Controller

**Run your application** and test the following actions:

* Add a new movie using the **Create** form. The movie should be stored in the database and displayed in the movie list
* Go to **/Movie/Index** to see a list of all movies retrieved from the database
* Click on a movie’s Details link to see detailed information about the selected movie

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer

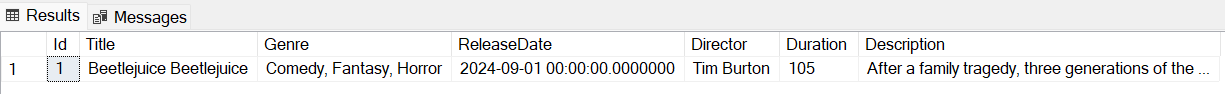
Description automatically generated

A screenshot of a movie

Description automatically generated

#### Check the Database in SSMS

* Open **SQL Server Management Studio (SSMS)**
* Navigate to the **Movies** table in your database and verify that the movie data is stored correctly
* Ensure that the movies being displayed on the **Index** page are the same as those in the database



#### Implementing Controller Summary

* We’ve updated the **MovieController** to interact with the **AppDbContext** to work with the actual SQL Server database
* Movies are now being retrieved, added, and managed directly from the database