**BCIT**

**Comp 4952 HCI for Application Development**

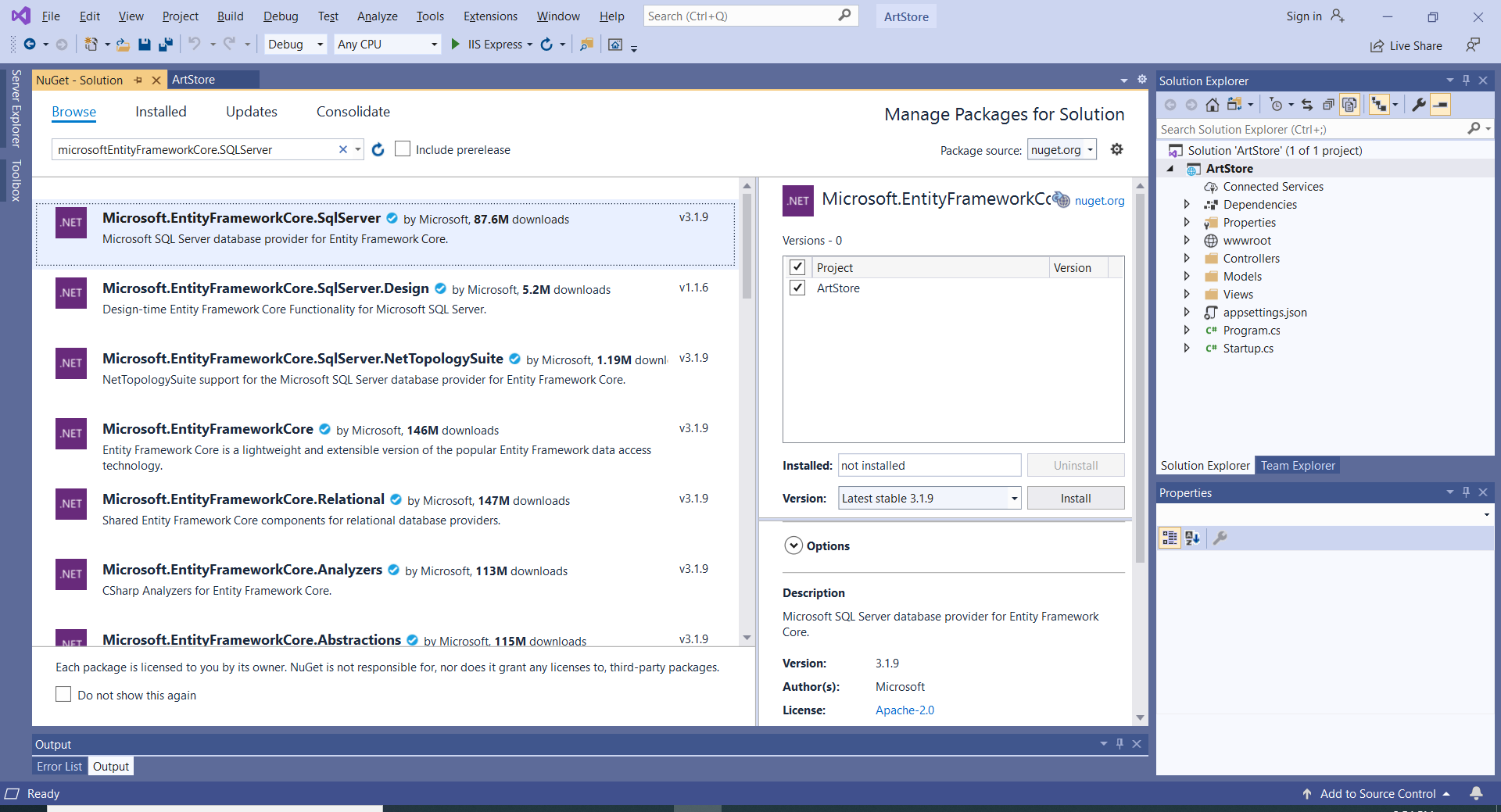
**Technical Programming Option**

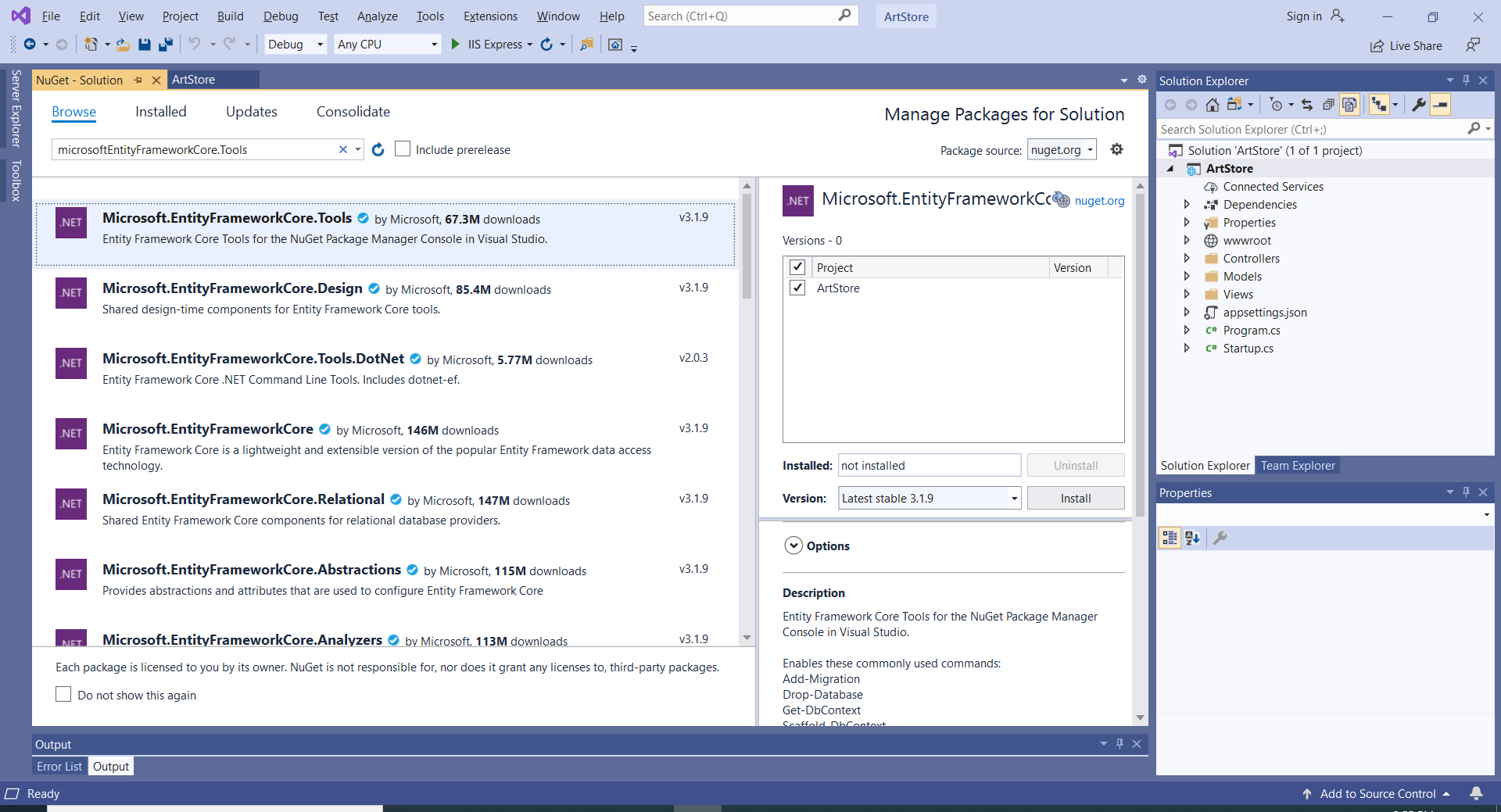
**Option Head Mirela Gutica**

**Fall 2022**

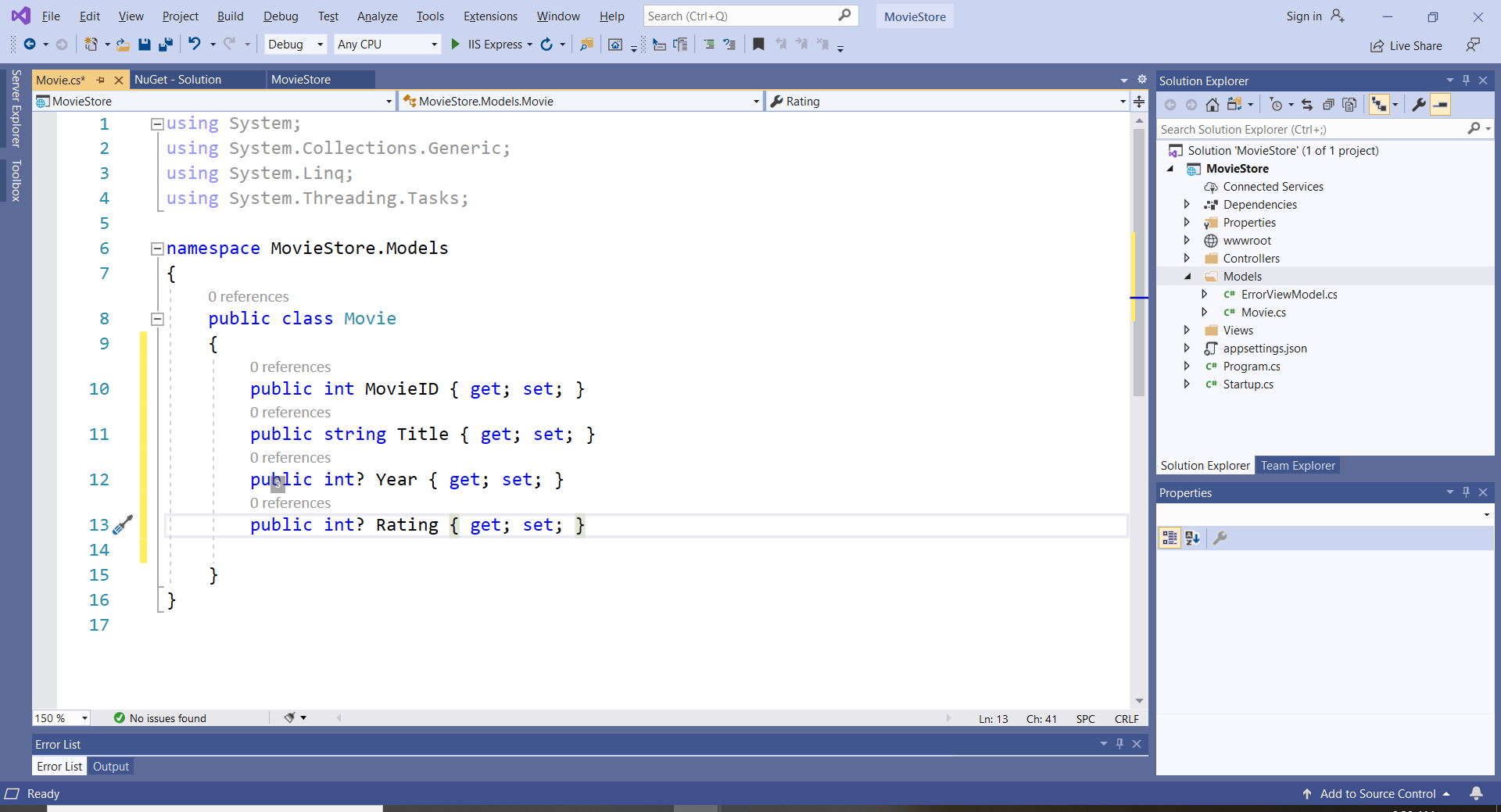
**Implement an Entity Framework Code-First ASP.NET Core MVC Application**

1. **Create an ASP.NET Core MVC Web Application**
2. **Install the EF and attach it to your project.** 
   1. Important: You need to add manually the EF to your project (as it is not automatically included with .NET Core 3.0 and later).
   2. Make sure that you use the right EF for your .NET Core Version that you have on your machine!
   3. Open the NuGet Package Manager.
      1. Select Tools/Nuget Package Manager/Manage NuGet Packages for Solutions
      2. Select Browse
      3. Select Microsoft.EntityFrameworkCore.SQLServer
      4. In the right-hand panel check your project and solution.
      5. Install
      6. Select Microsoft.EntityFrameworkCore.Tools
      7. In the right-hand panel check your project and solution.
      8. Install

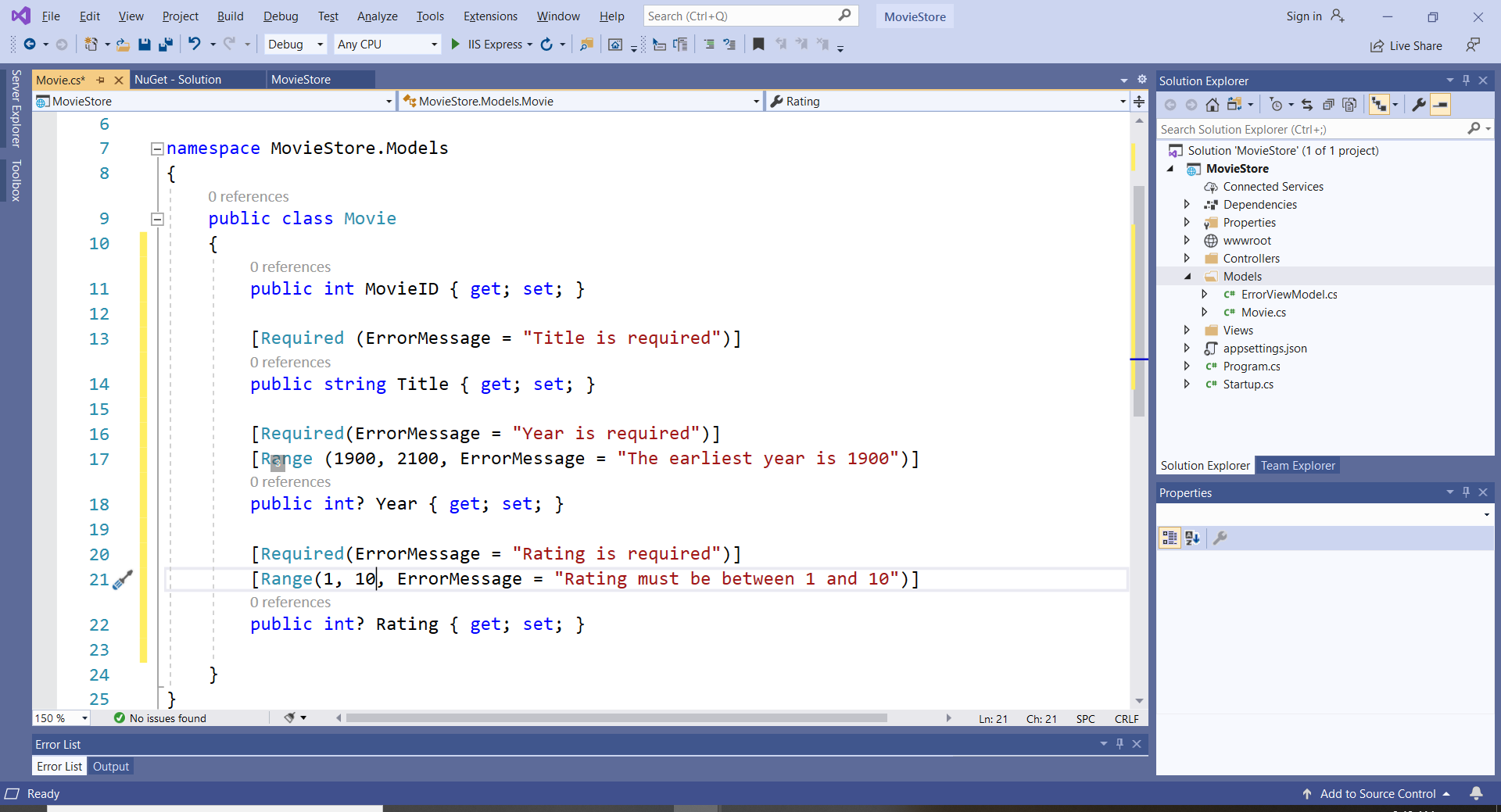




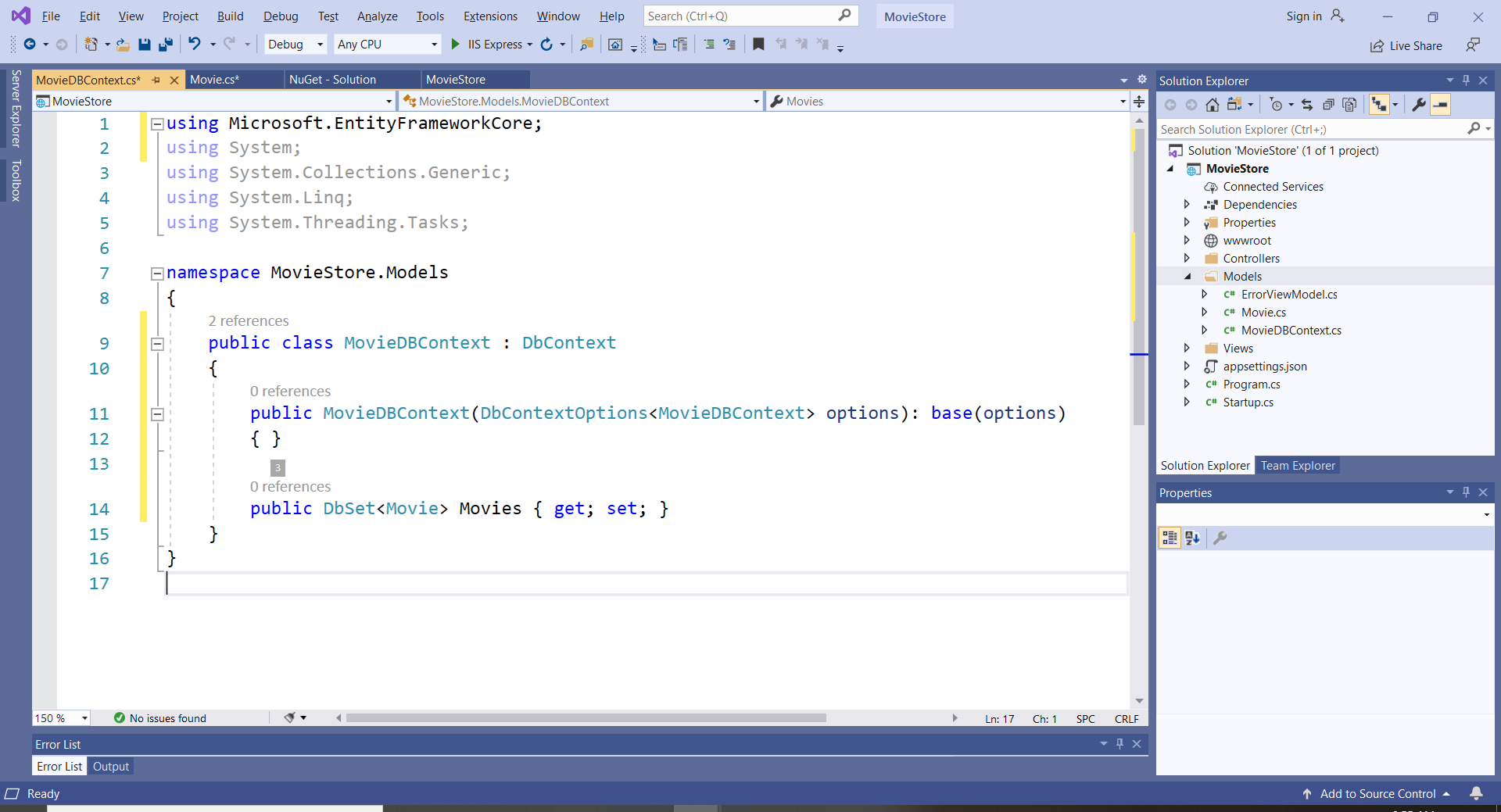
1. **Create entities**
   1. Right click on the Model folder and add a new item class.
   2. Implement the class.
   3. The MovieID will allow the generation of a primary key in the DB.
      1. Any property with a name of Id or ID or the entity name followed by Id or ID is a primary key.
      2. If a property is of int type is an identity column whose value is automatically generated.



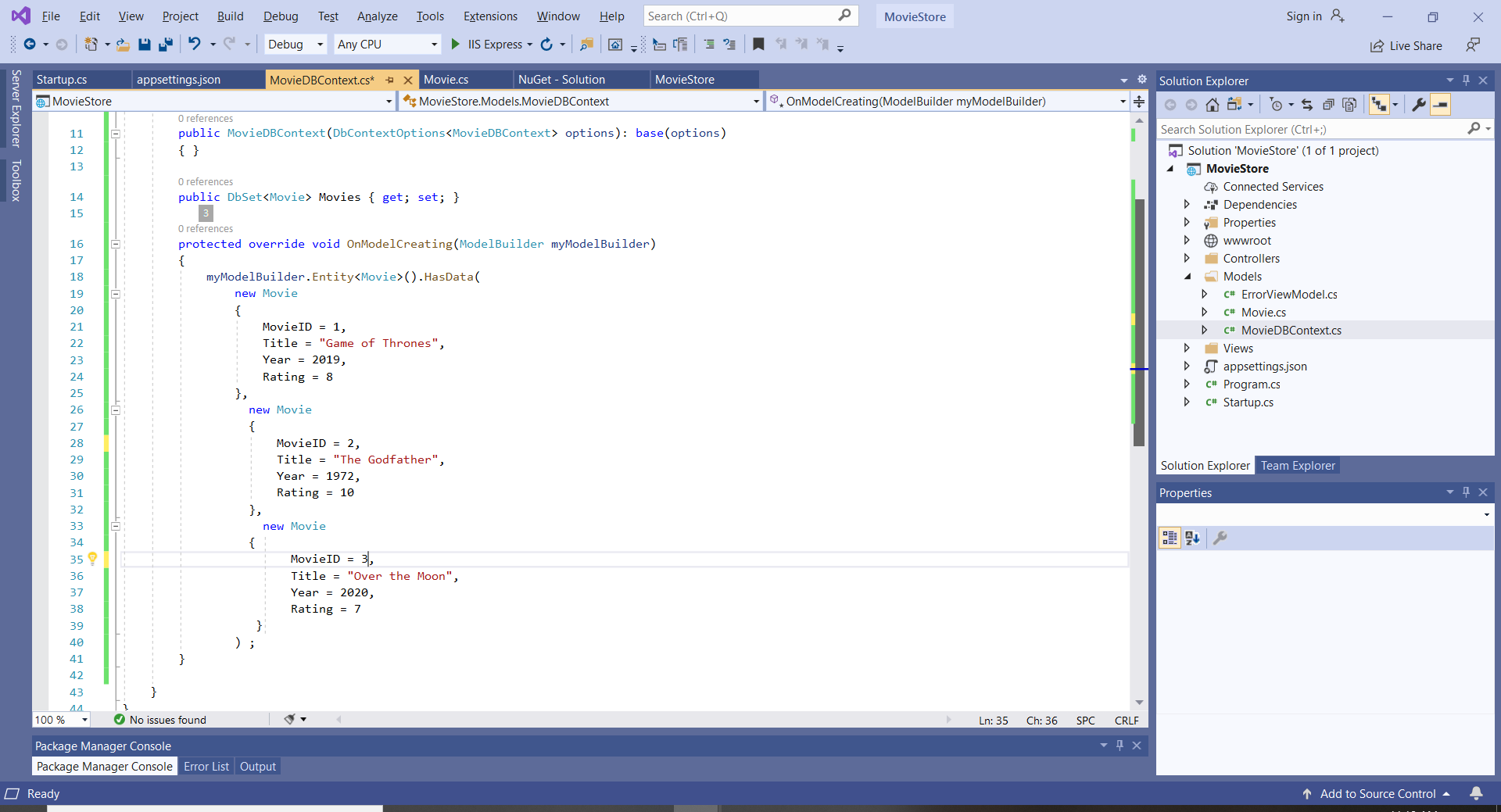
* 1. Add validation



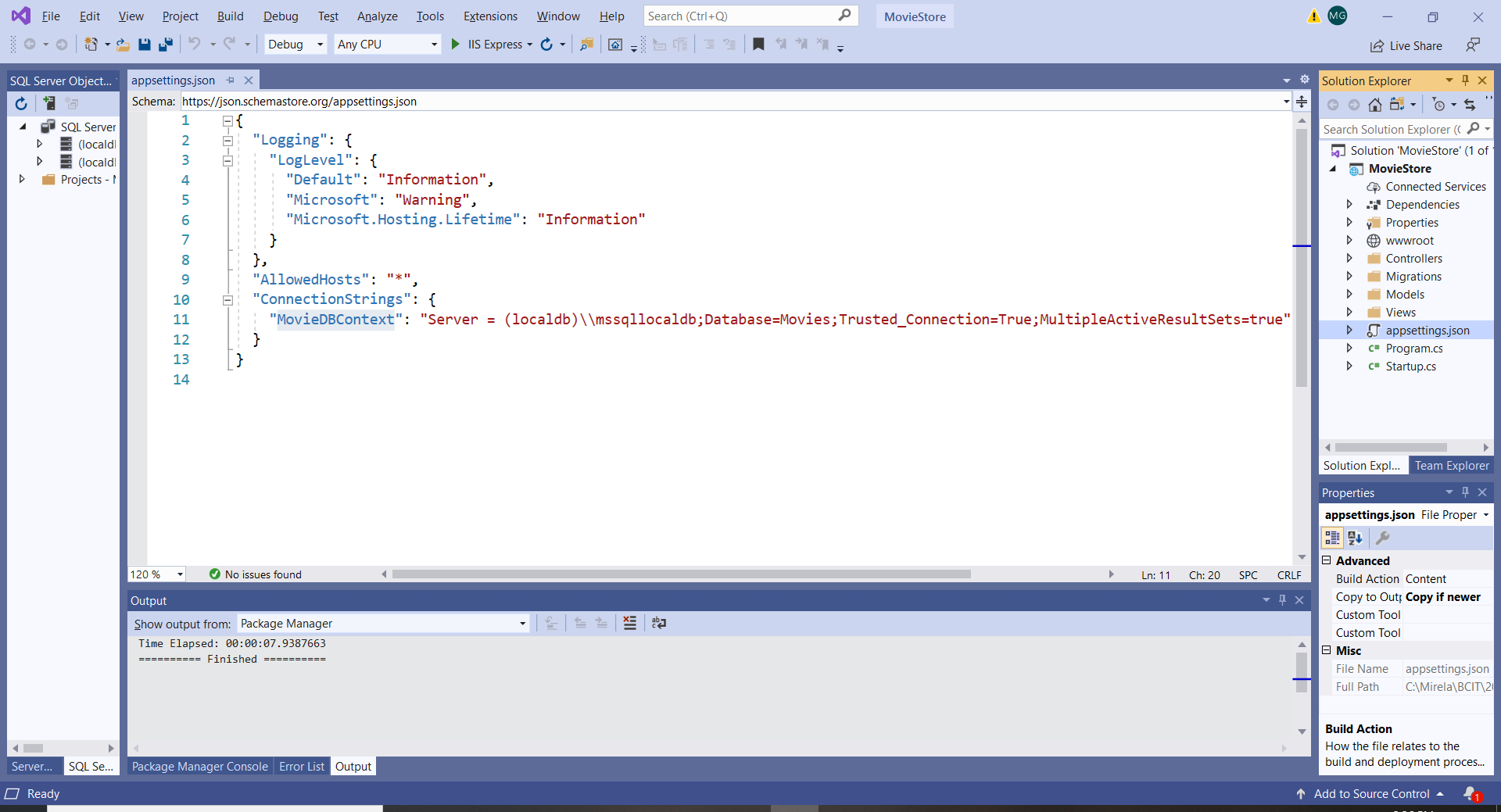
1. **Create a DB Context class.**
   1. The MovieDbContext class should be created in the Model folder.
   2. Create a MovieDbContext class that inherits DbContext base class.
   3. The DBContext which is part of the Microsoft.EntityFrameworkCore namespace, allows communication with the DB.
   4. Add the property DbSet<Entity> type. The DbSet will allow the generation of the DB.



1. **Seed initial data**
   1. This step is not required, but it could be useful for testing or to generate some initial data
   2. The data is seeded in the DbContext class.
   3. The OnModelCreating() method accepts a ModelBuilder object.



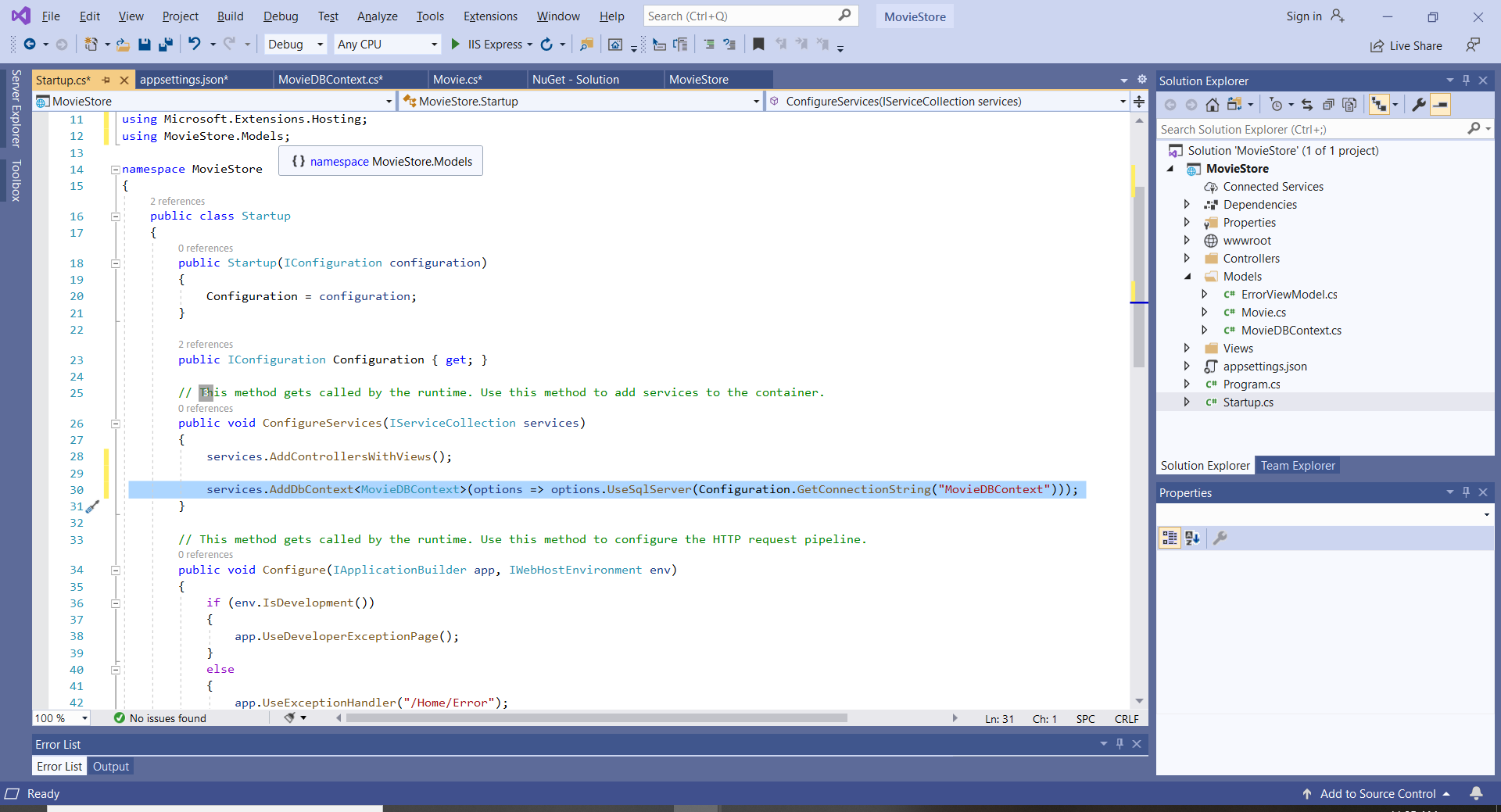
1. **Add a “connection string” to connect to the DB**
   1. The connection string should be added in the configuration file appsettings.json



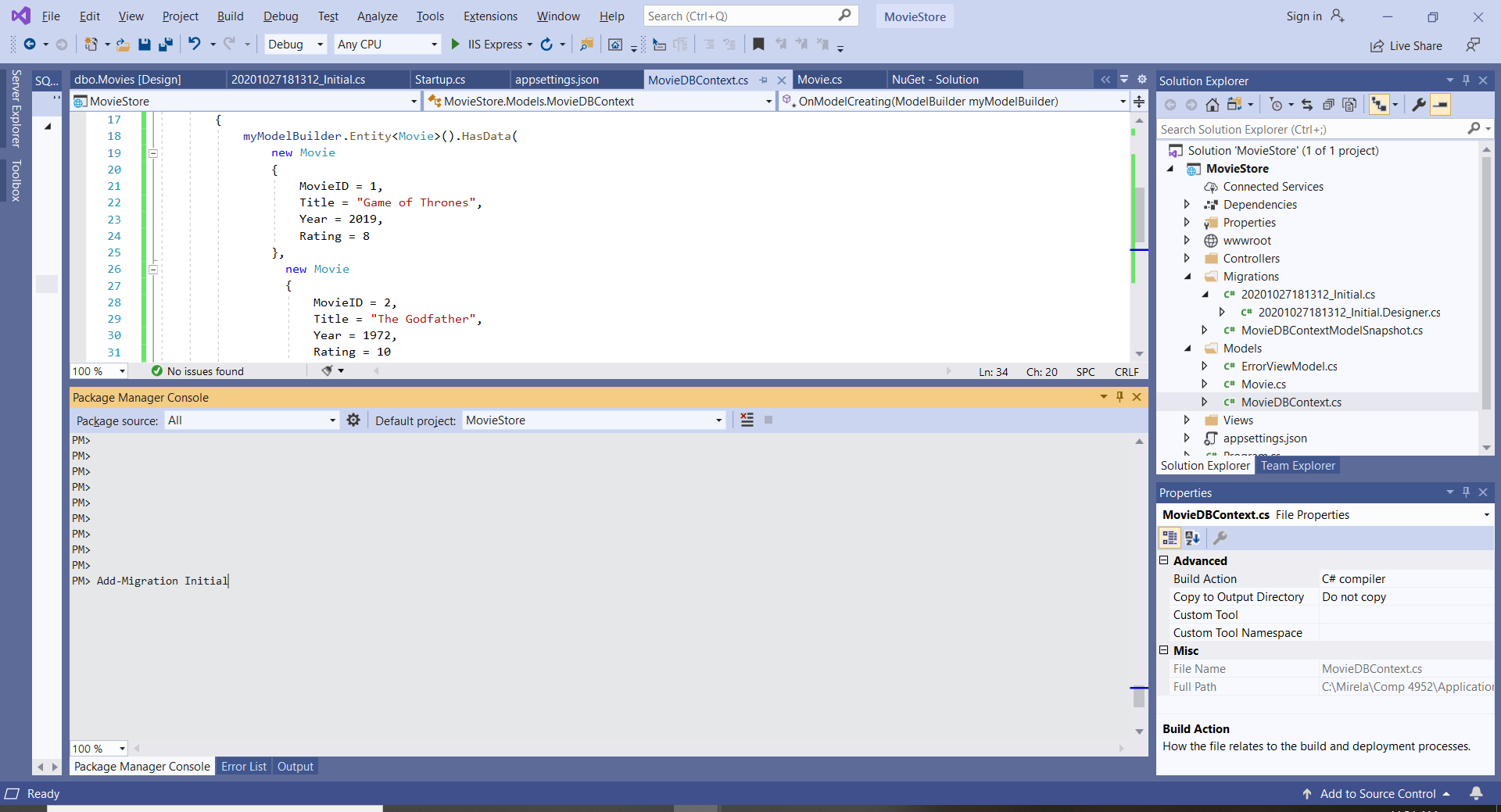
1. **Enable dependency injection**
   1. Core MVC uses dependency injection to pass the DbContext objects to the controllers
   2. The dependency injection should be added in the Startup file
   3. Make sure that you add the Model namespace to the Startup file.

using Microsoft.EntityFrameworkCore;

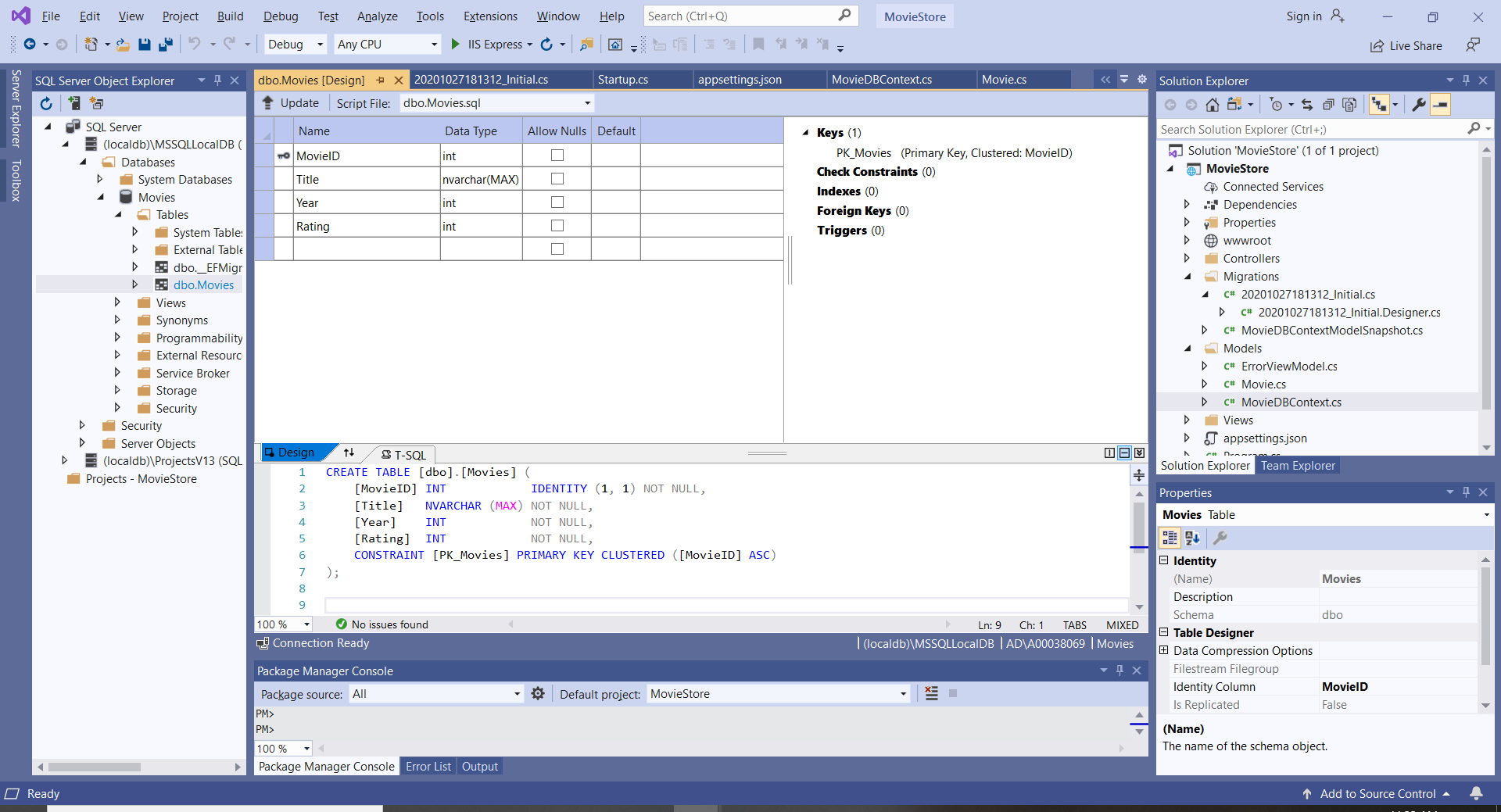
using MovieStoreApp.Models;



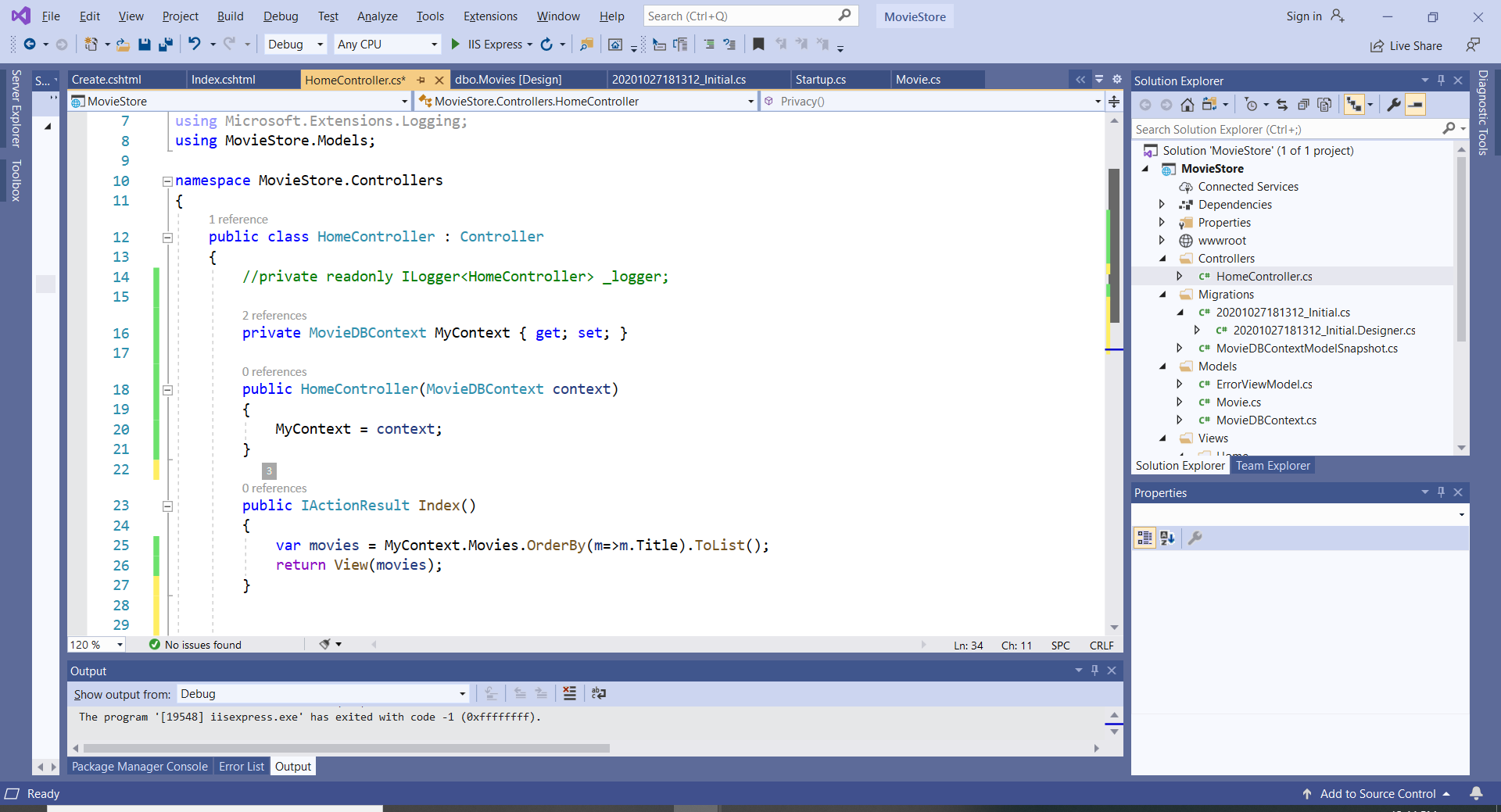
1. **Create the DB**
   1. Select Tools -> NuGet Package Manager -> Package Manager Console Command
   2. In the PM console give the commands:
      1. “Add-Migration Initial” and press Enter
      2. “Update-Database” and press Enter



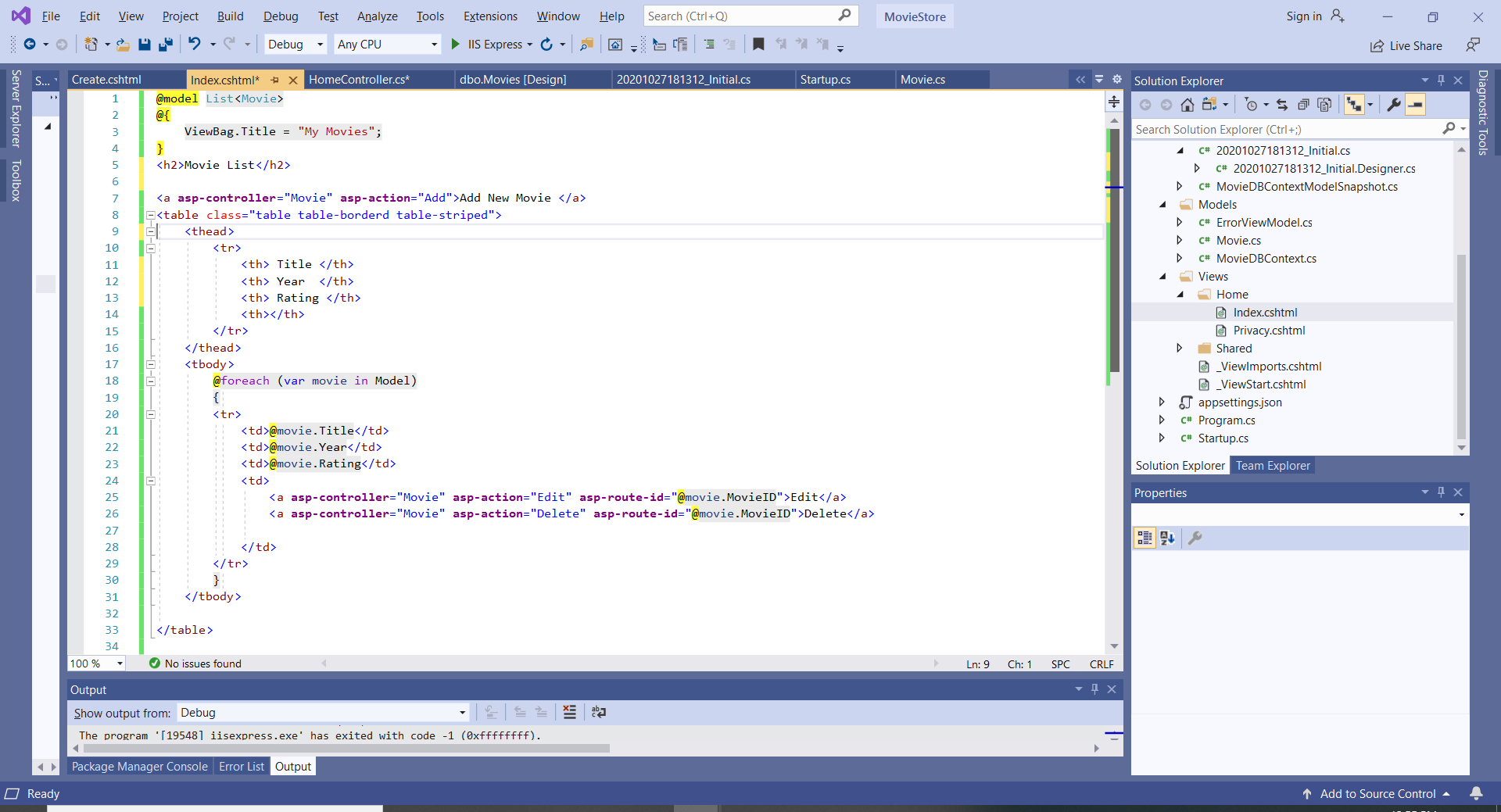
* 1. The database will be created
  2. You can see the Movies database in the localdb



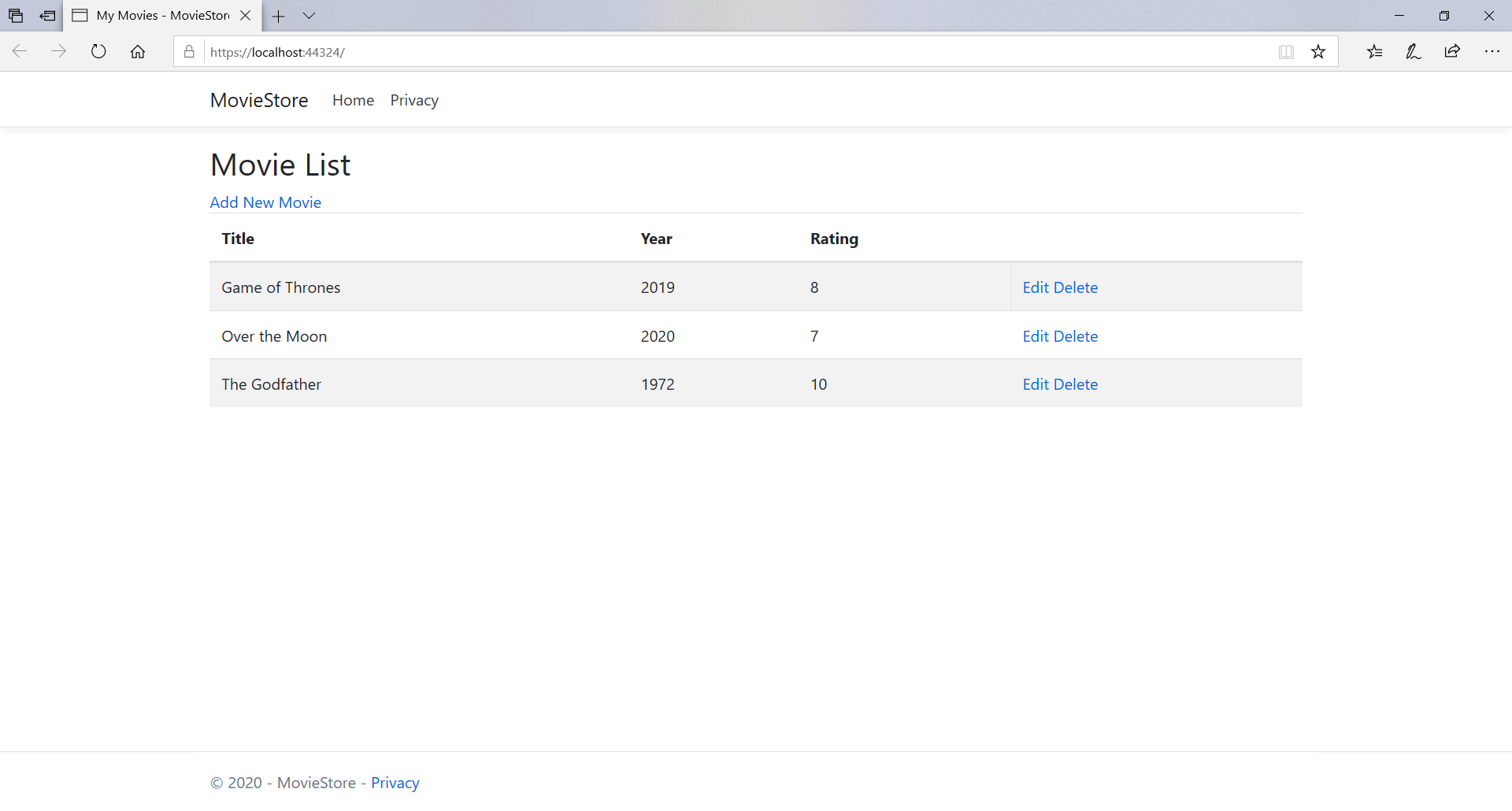
1. **Add/Update the Home Controller**
   1. Create a “context” private property of type MovieDBContext. The constructor accepts the MovieDBContext object and assigns it to the context property.
   2. In the index action the context property is used to get the collection of movies from the DB.
   3. The collection of movies is passed to the DB.



1. **Add/Update the Home View**
   1. The Home View should be updated to display the movie list.
   2. In the Home/Index view add the HTML code
   3. Use the ASP.NET Core MVC tag helpers “asp-controller” and “asp-action”



1. Run the Web Application:

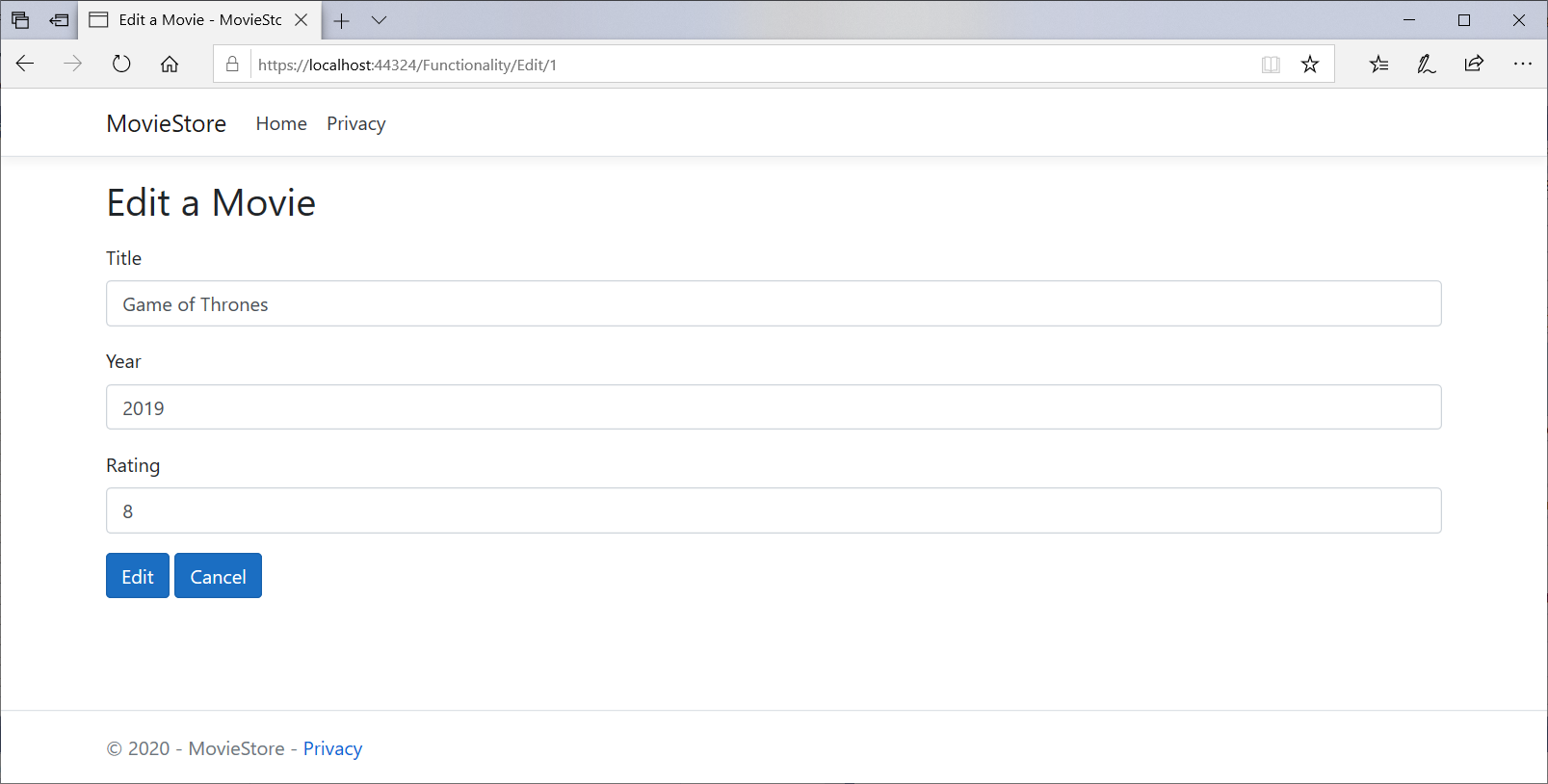


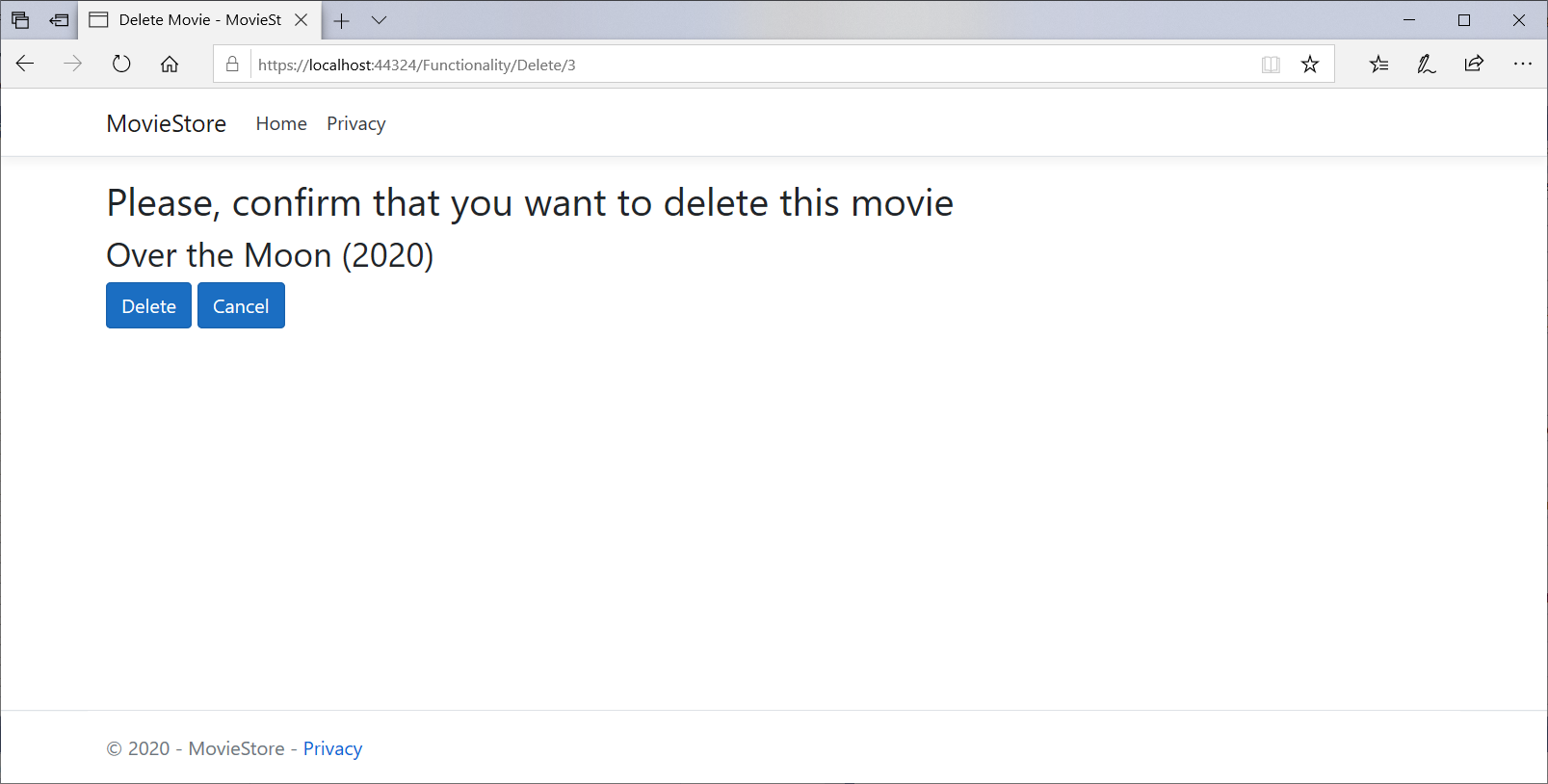
1. **Please observe the file and folders in the created ASP.NET Core MVC application:**



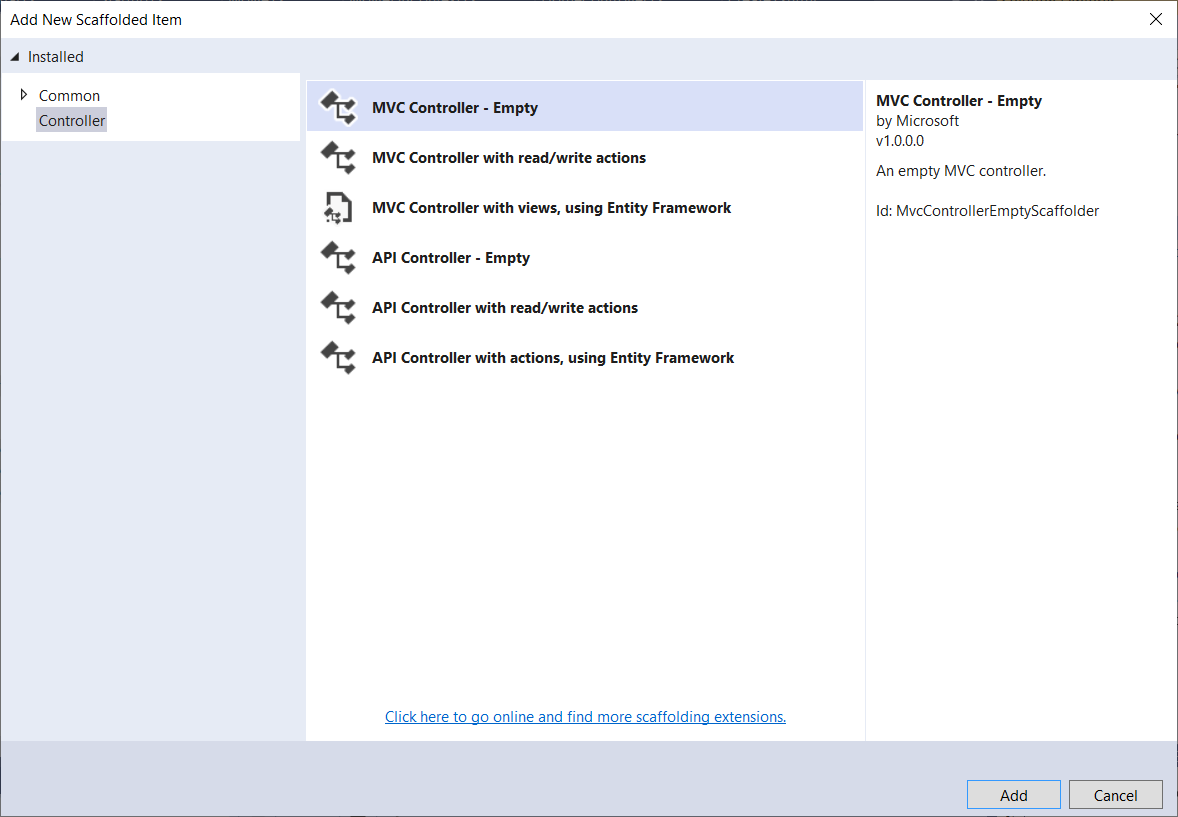
1. **Add Functionality: Add, Edit and Delete**



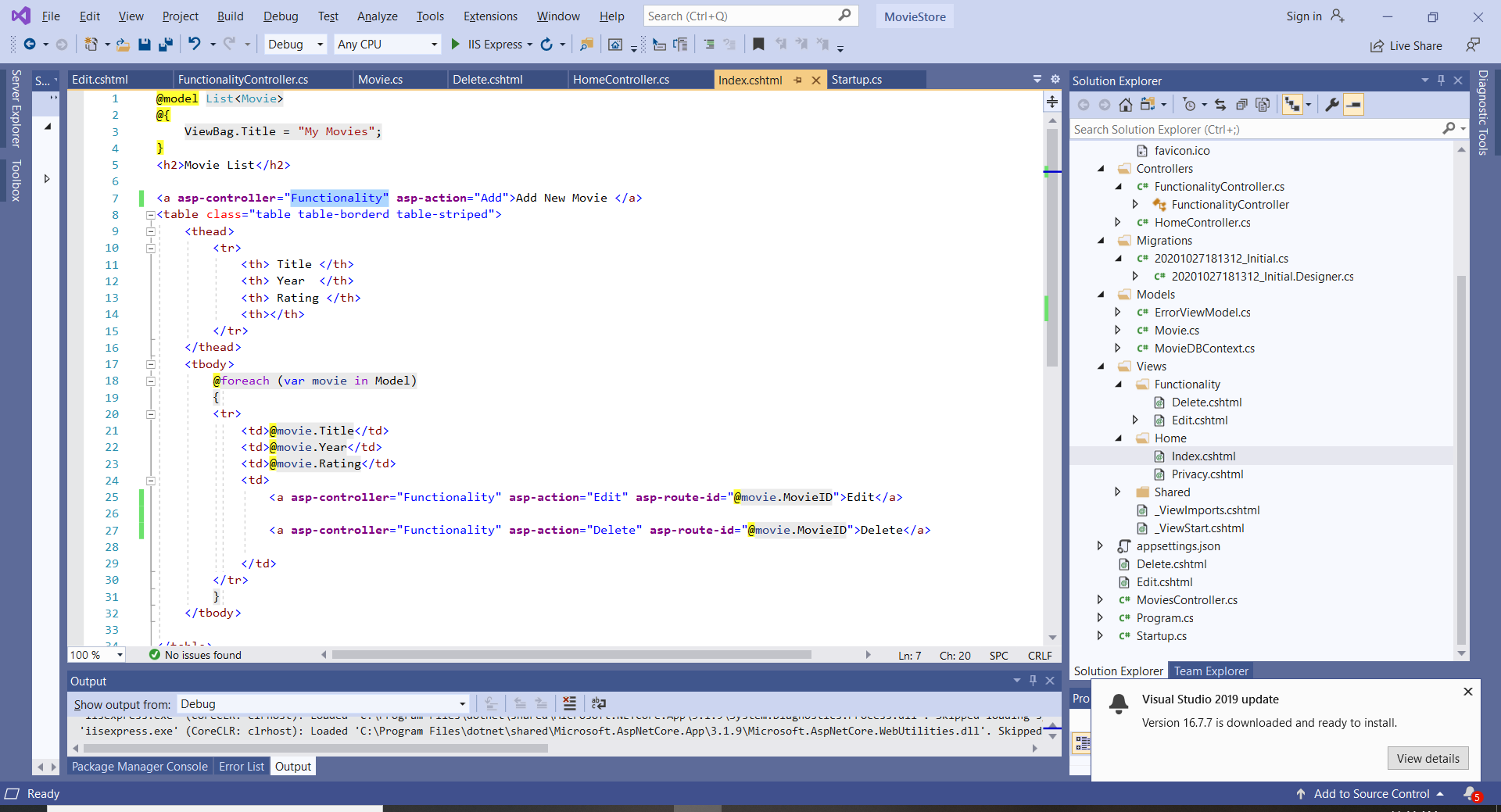




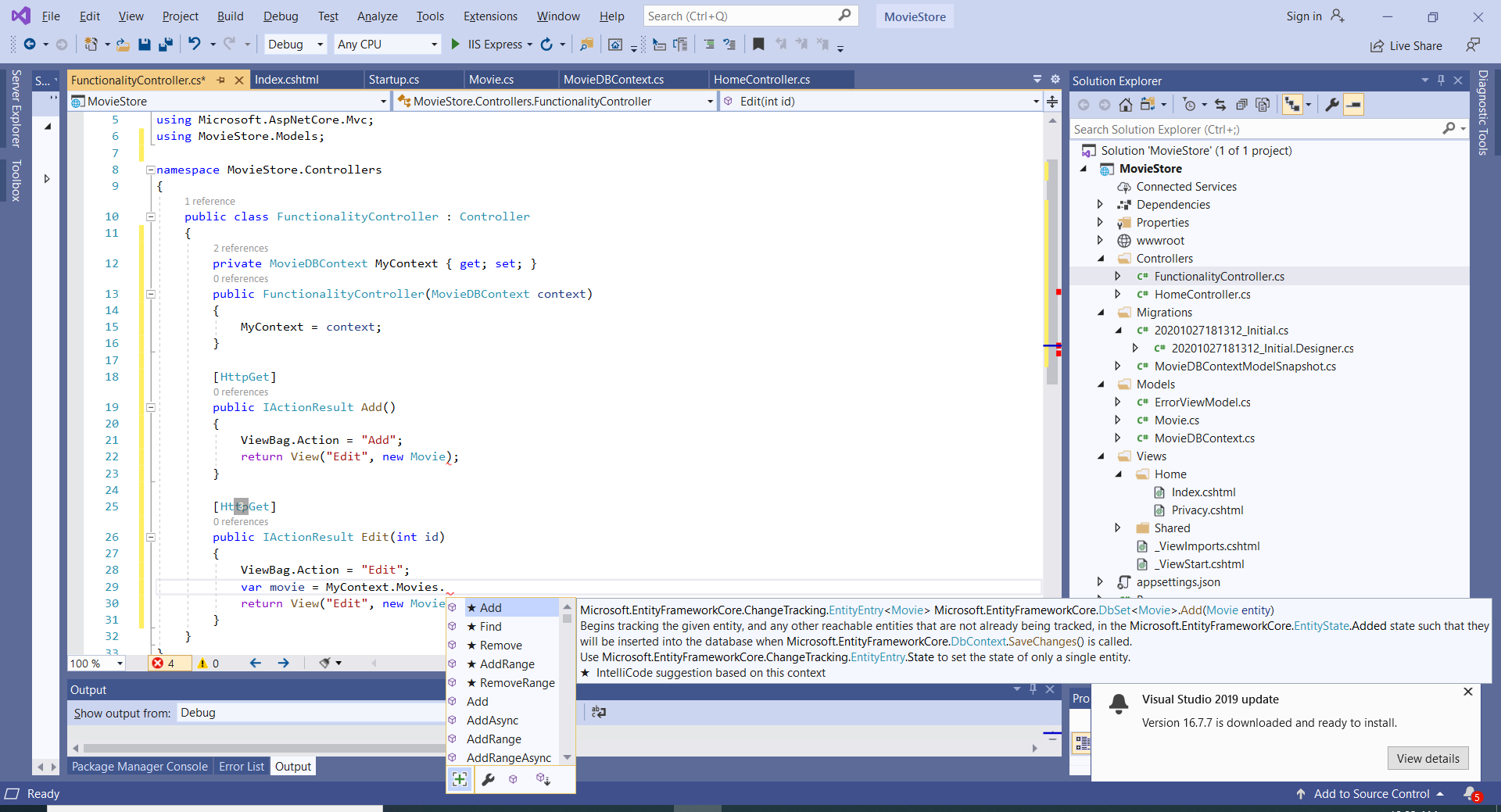
* 1. Add a Controller that executes the “Add New Movie”, “Edit” and “Delete” functionality
  2. Right click on the Controller folder and “Add” a Controller. Select an empty controller.



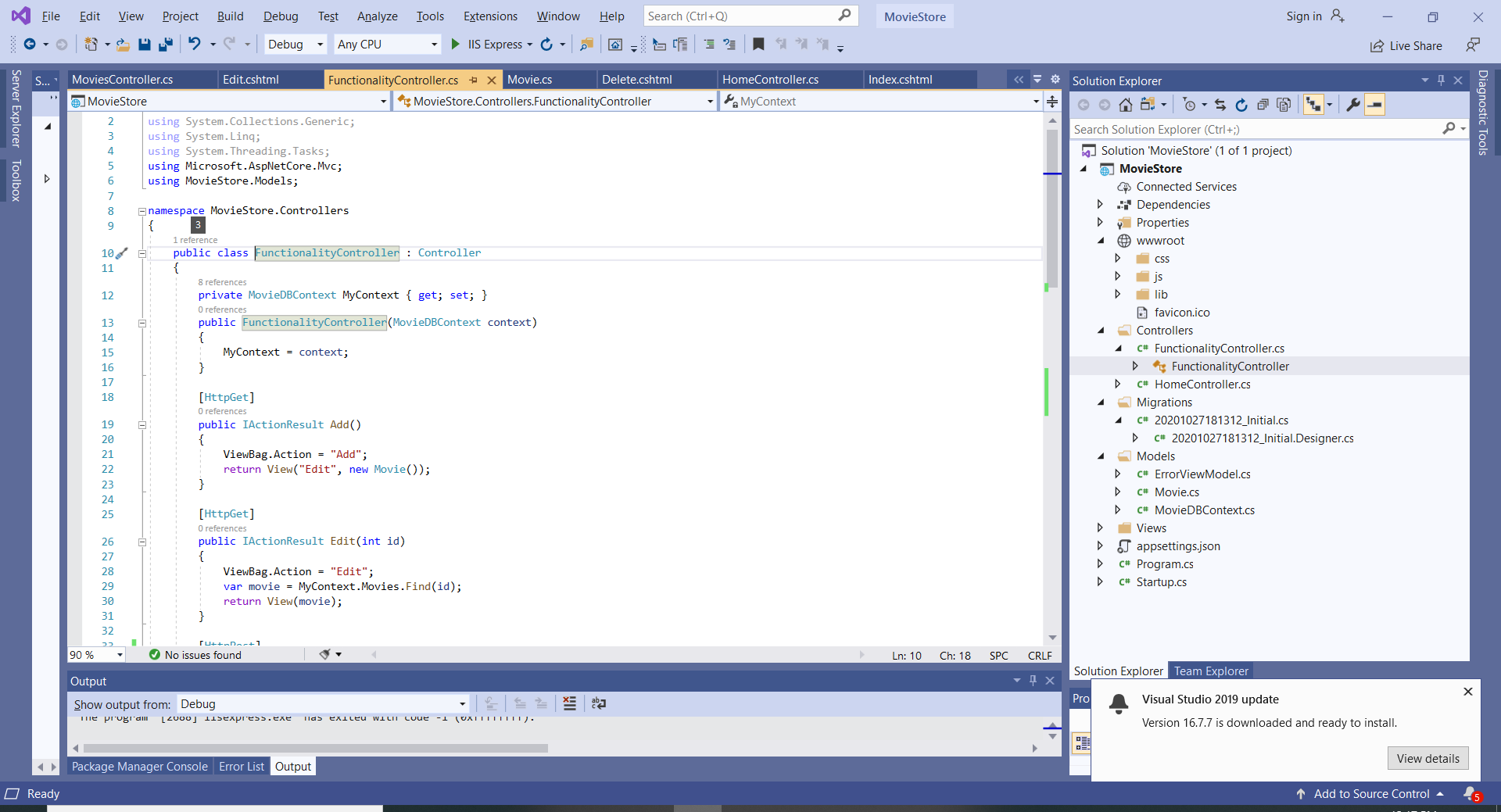
* 1. Name the controller with the name of your choice. I used the name “FunctionalityController”. Make sure that you update the name in the Home Controller (where a more general name “Movie” was used). Now the “Add New Movie”, “Edit” and “Delete” links are connected to the FunctionalityController.

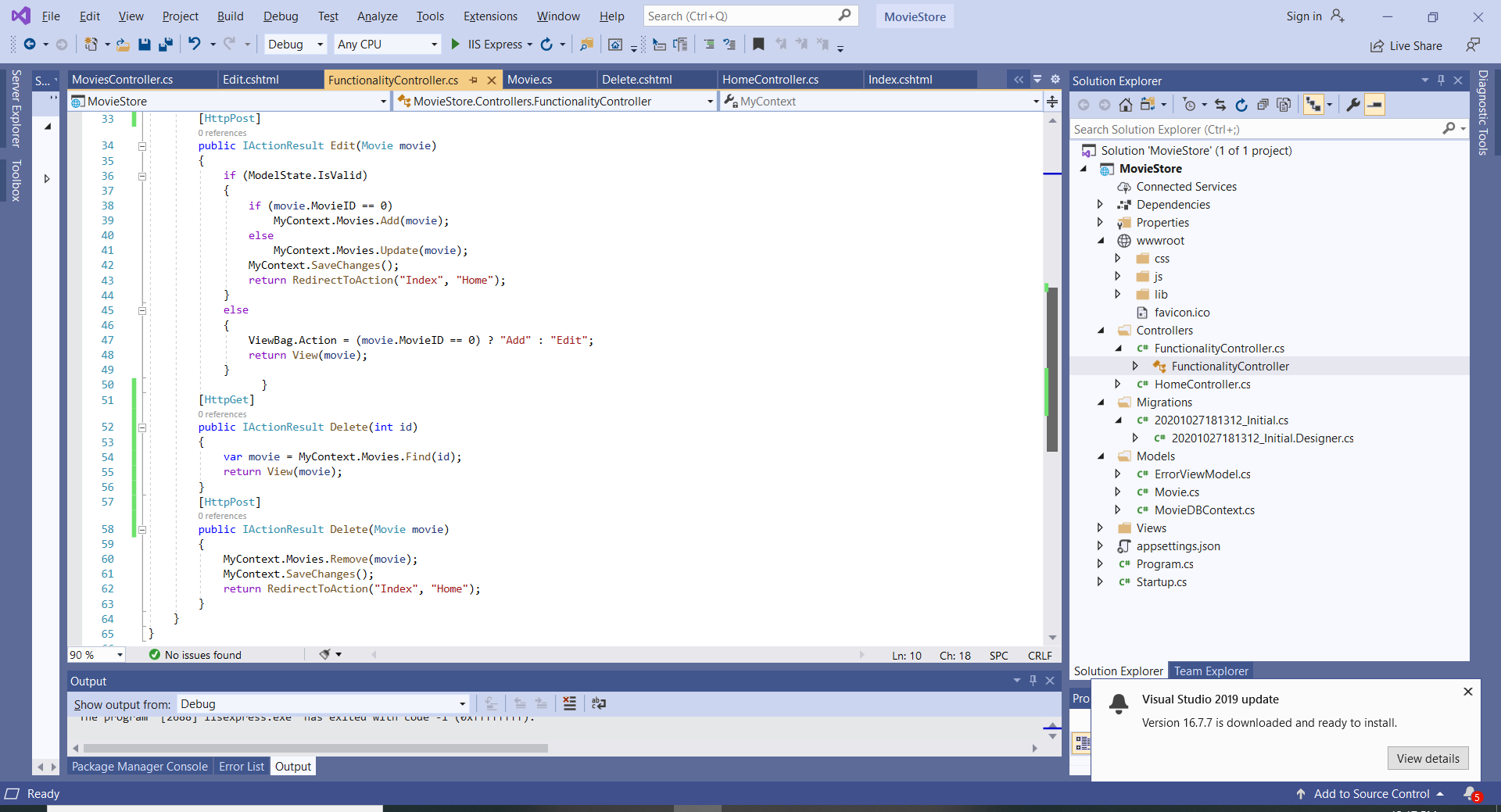
“ 

* 1. Similarly to the Home Controller, as you are using the Movie entity, create a “context” private property of type MovieDBContext. The constructor accepts the MovieDBContext object and assigns it to the context property.
  2. This controller has 3 actions that correspond to user’s requests: Add, Edit and Delete. “Add” handles only POST requests (there is no need for GET – do you see why?). “Edit” and “Delete” handle both POST and GET requests.
  3. Note that when you are using the entity (Movies in this case), several methods are already implemented (e.g., Add, Find, Remove, etc.). Why?



* 1. Note that for GET requests, the View is returned. For POST requests, if the action is successful (not canceled), the context is saved and the action is redirected to the Home controller.





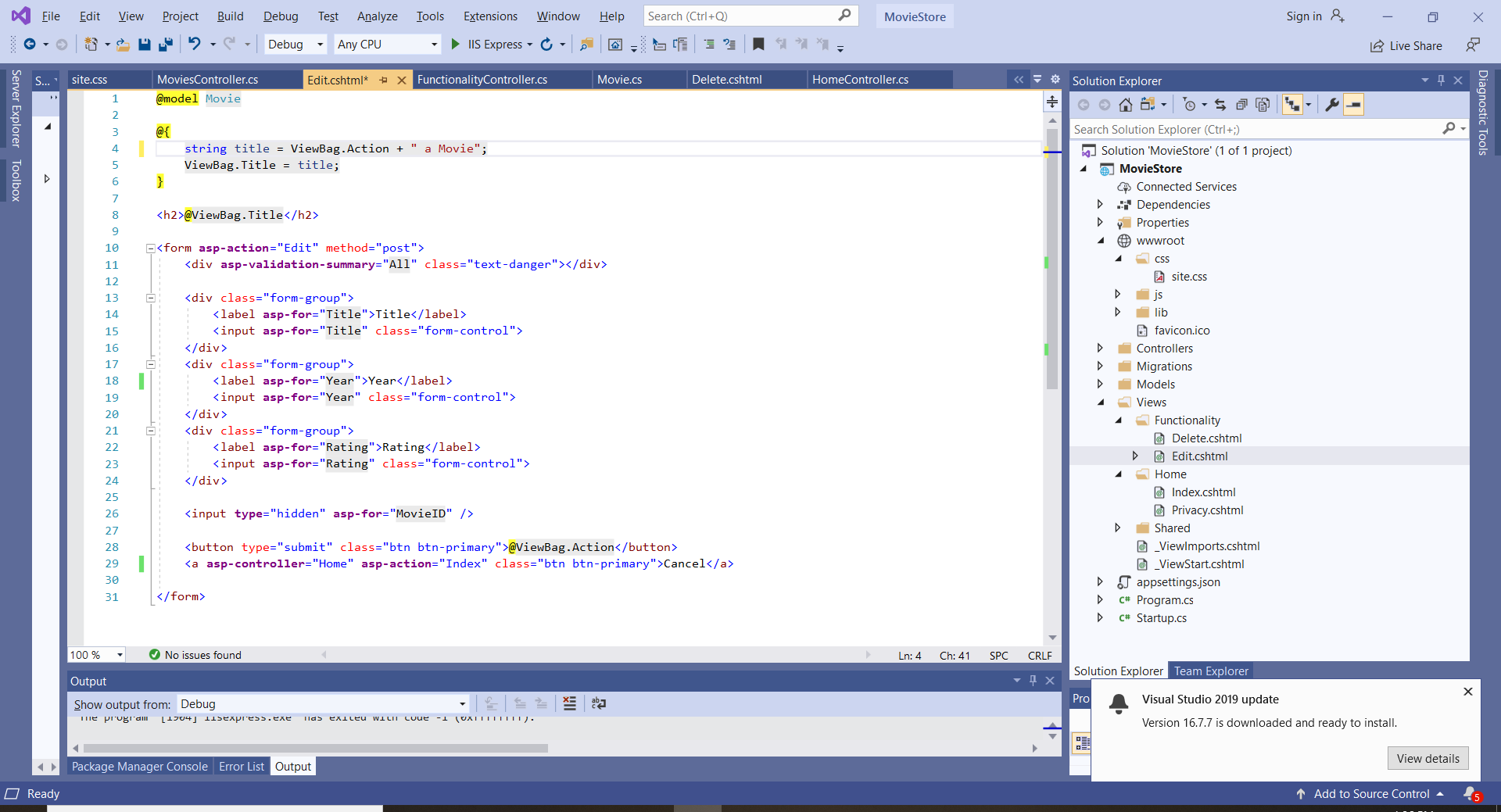
* 1. In the View, add a view folder named the same as the controller (in this case Functionality) and inside this folder add two empty Razor Views: Edit and Delete. There is no need for an Add View as the Edit View is used for Add New Movie, too.



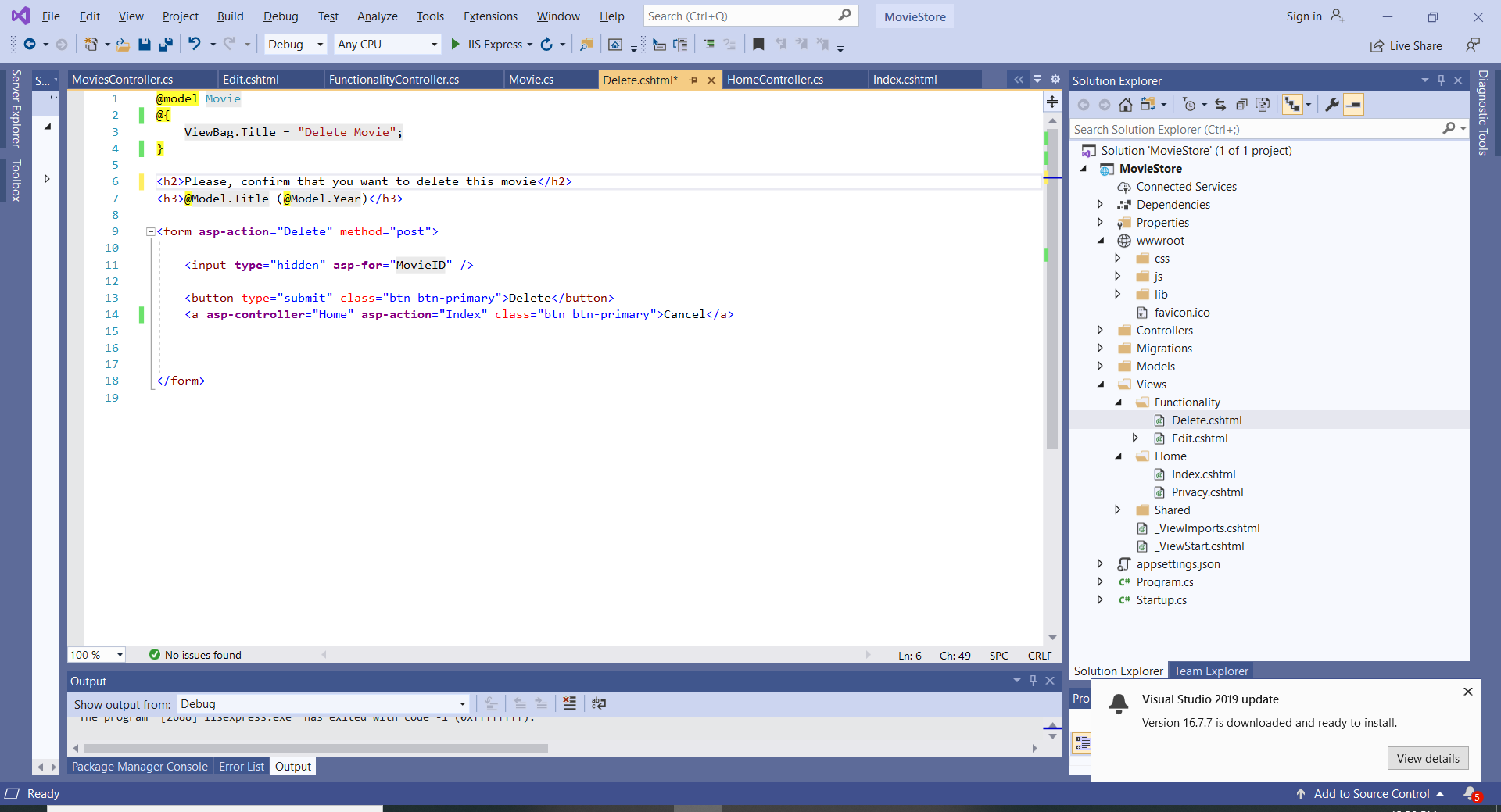
* 1. Here is a possible implementation of the Edit View.
  2. Note that the action Add New Movie or Edit is passed in the ViewBag:

string title = ViewBag.Action + " a Movie";

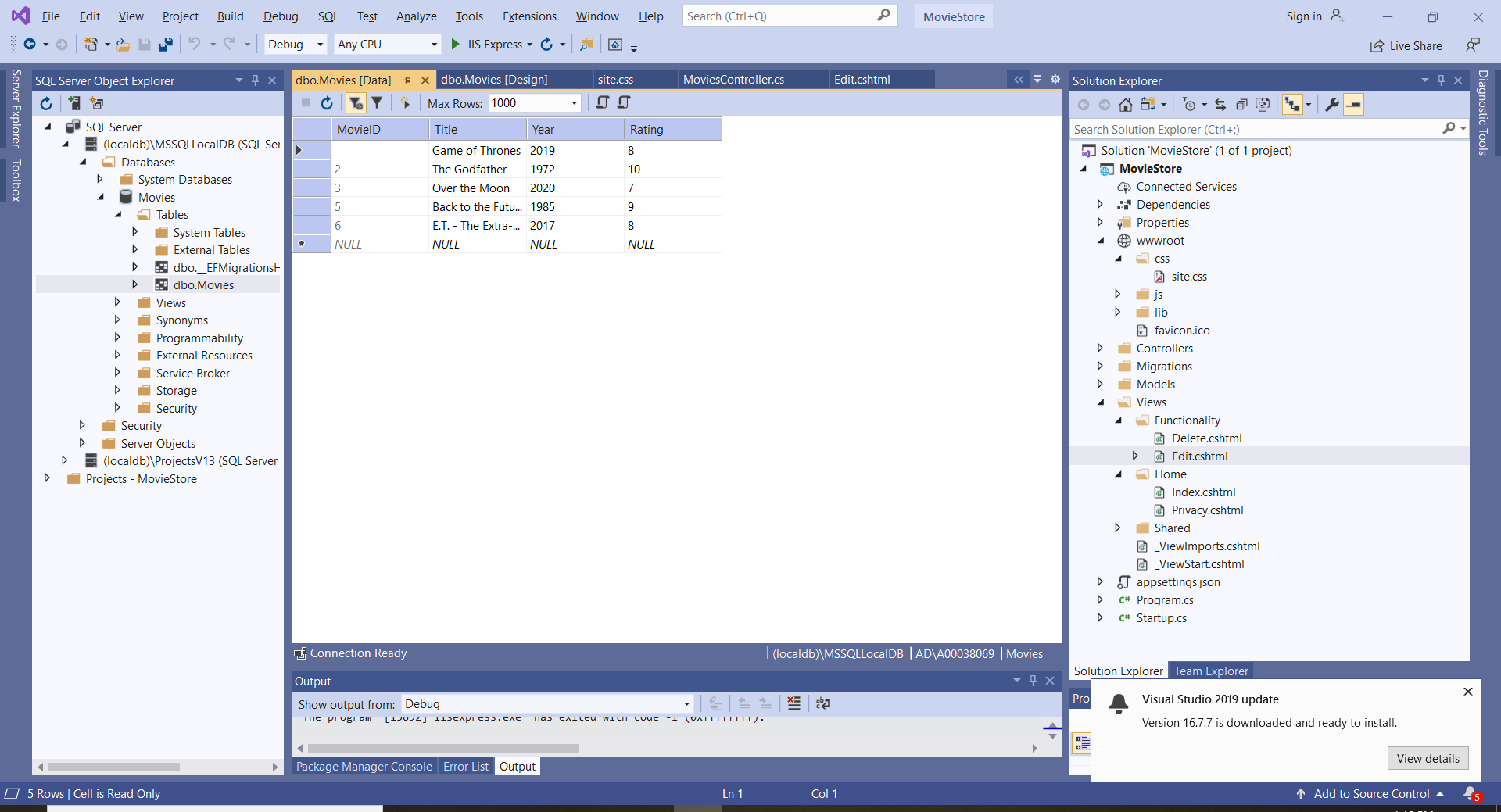
* 1. Note the use of ASP.NET Core MVC tag helpers:



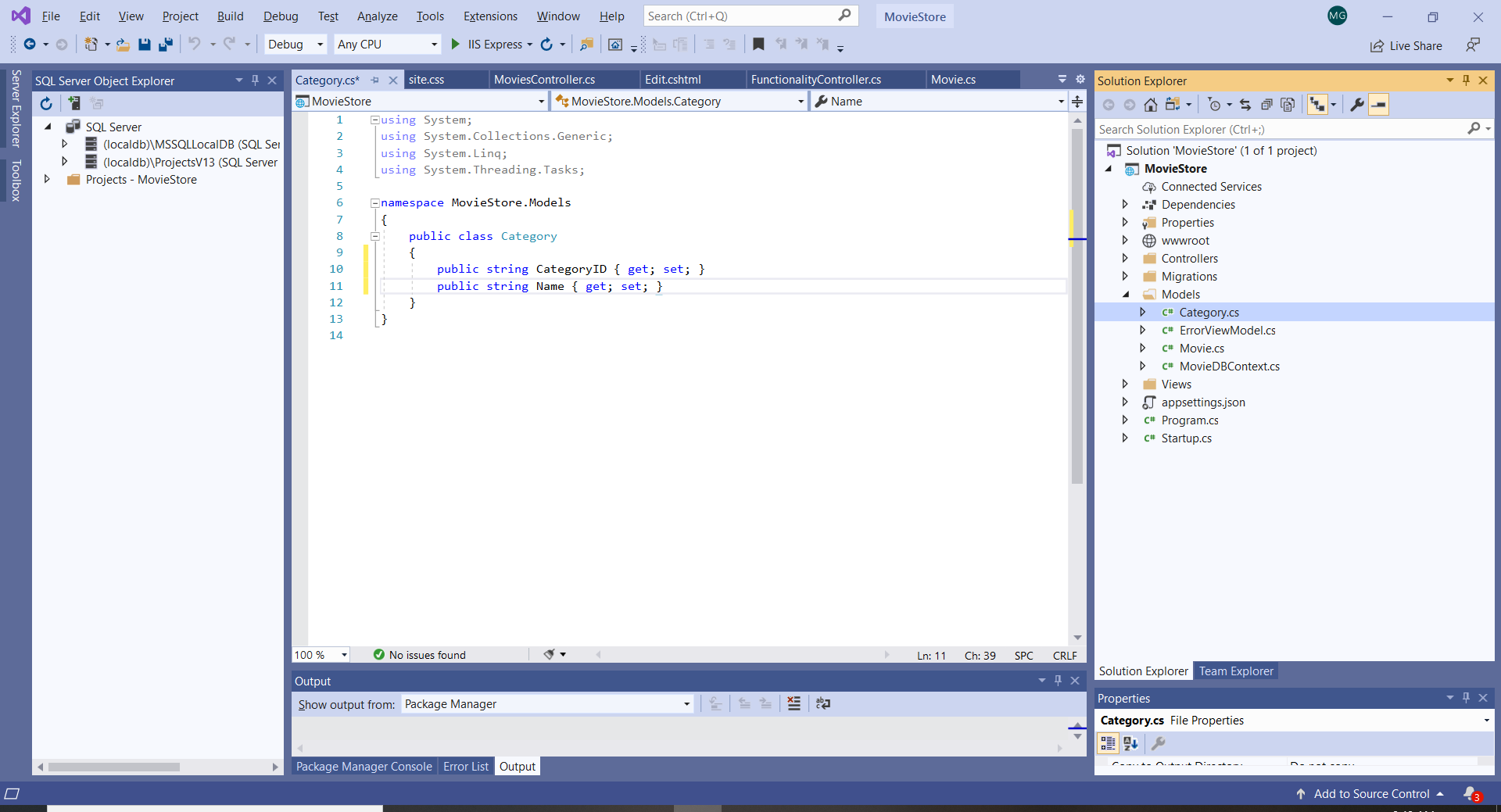
* 1. The last two lines add a “submit” button that facilitates the action to be performed, and a Cancel link that allow the return to the Home Controller with no action.
  2. Note that the MovieID is bound to a hidden field. Therefore the user cannot change this value.
  3. Here is a possible implementation of the Delete View:



* 1. The Delete View posts to the Delete() action, and similarly, Cancel allows the return to the Home controller with no action.
  2. Run and test the Application
  3. Observe the DB being updated:

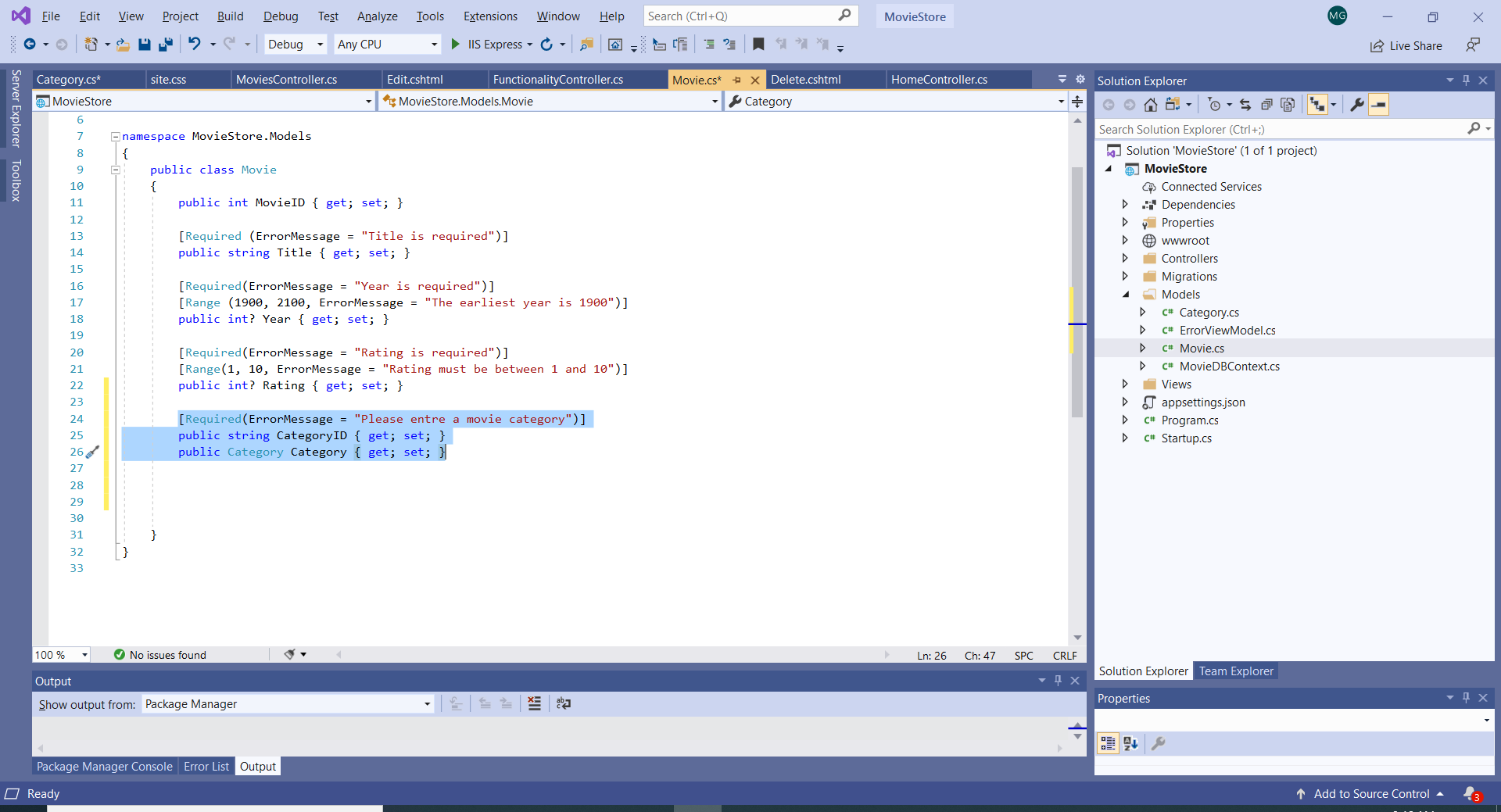


1. **Adding More Entity Classes and Relating Entities**
   1. Add a new Entity Class “Category”. This class has two properties: CategoryID (primary key) and Name.
   2. Add a new Model named Category.



Note that the primary key is of “string” type, therefore the DB will not generate values for it.

* 1. In order to relate the two entities, add a property Category to the Movie entity and a foreign key property.



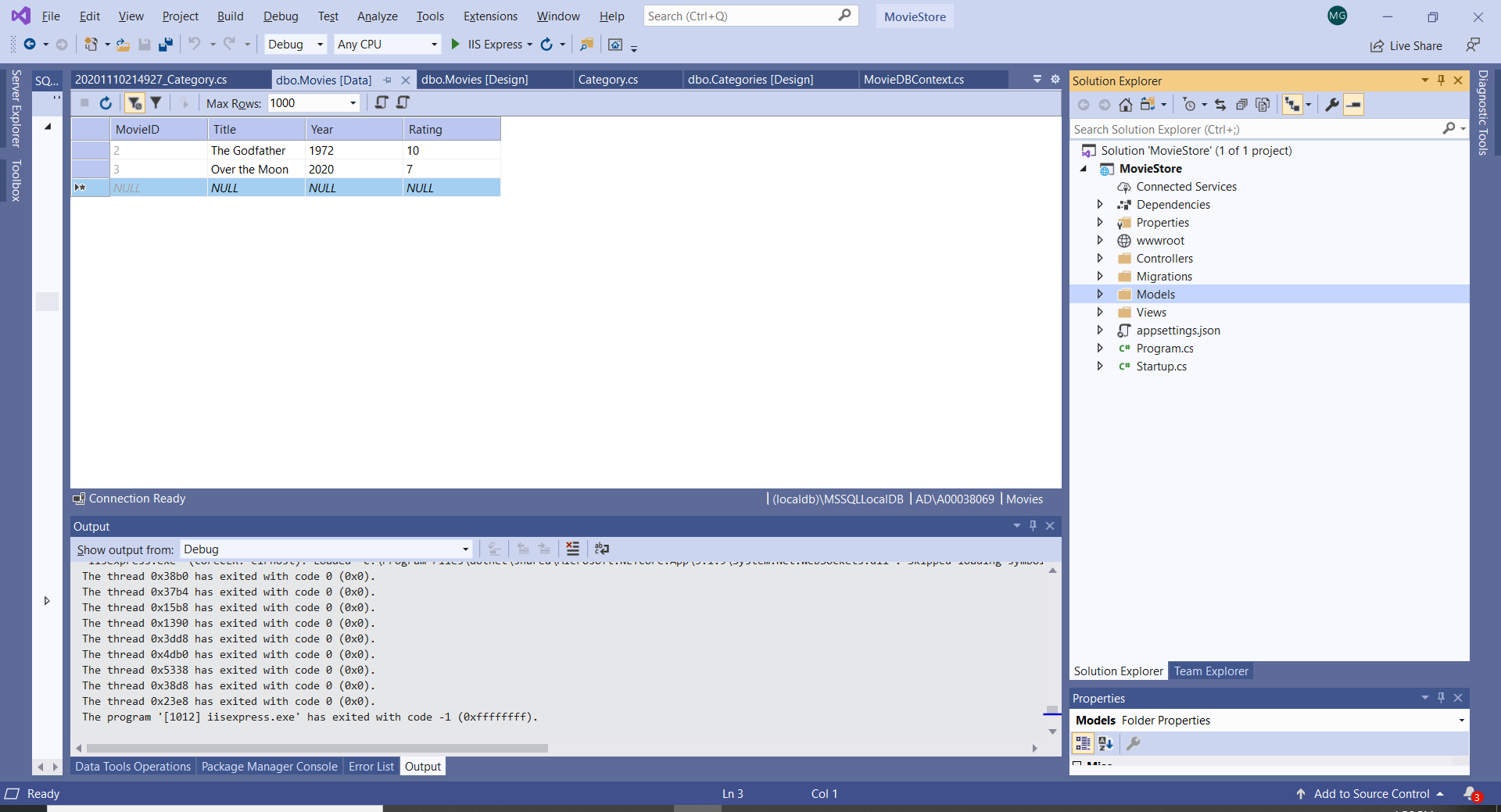
Movies and Categories are entities that are related. The foreign key is a key that is a primary key in the related Category class.

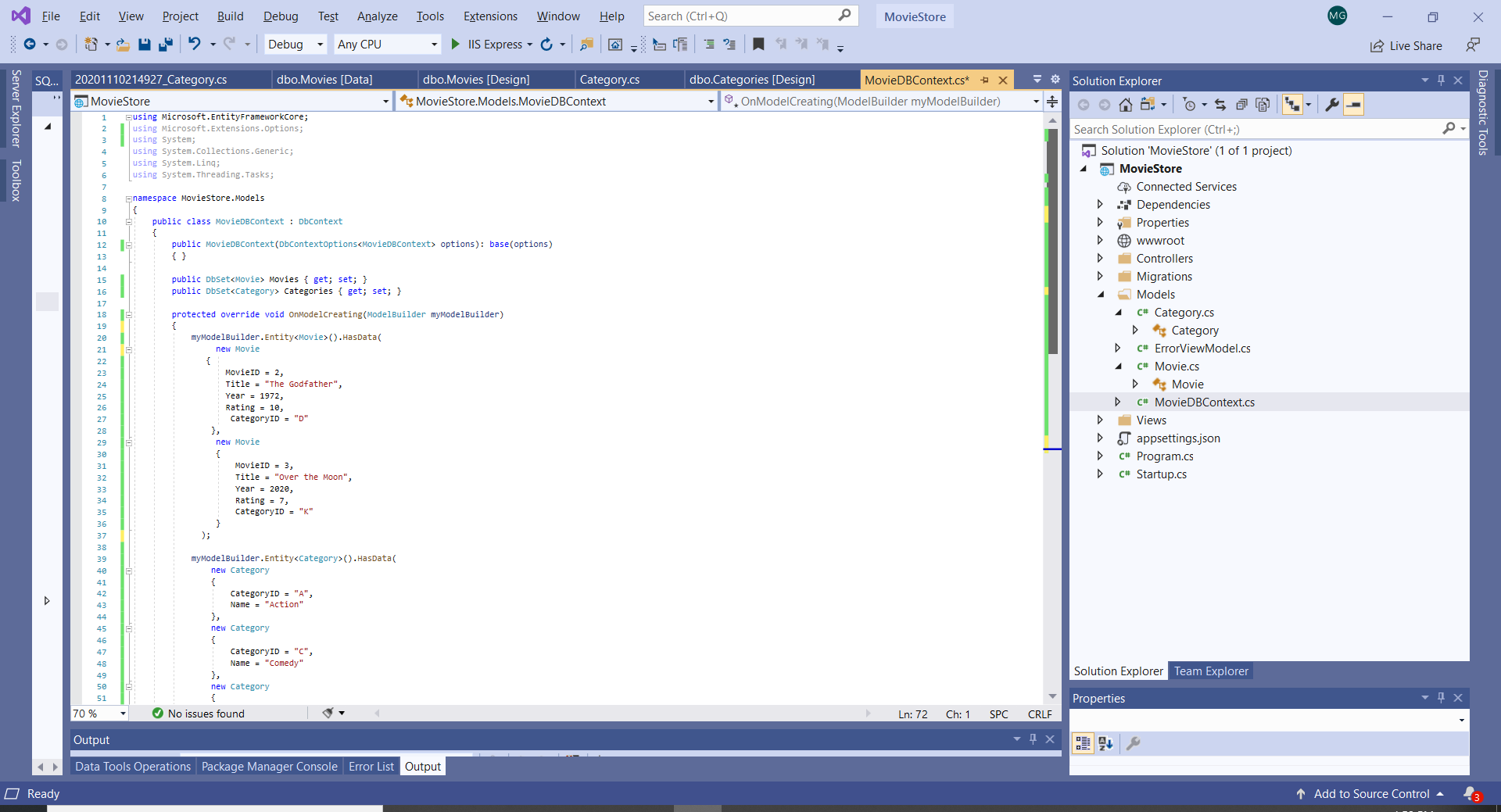
Foreign key are is used to seed, update, query and validate the related data.

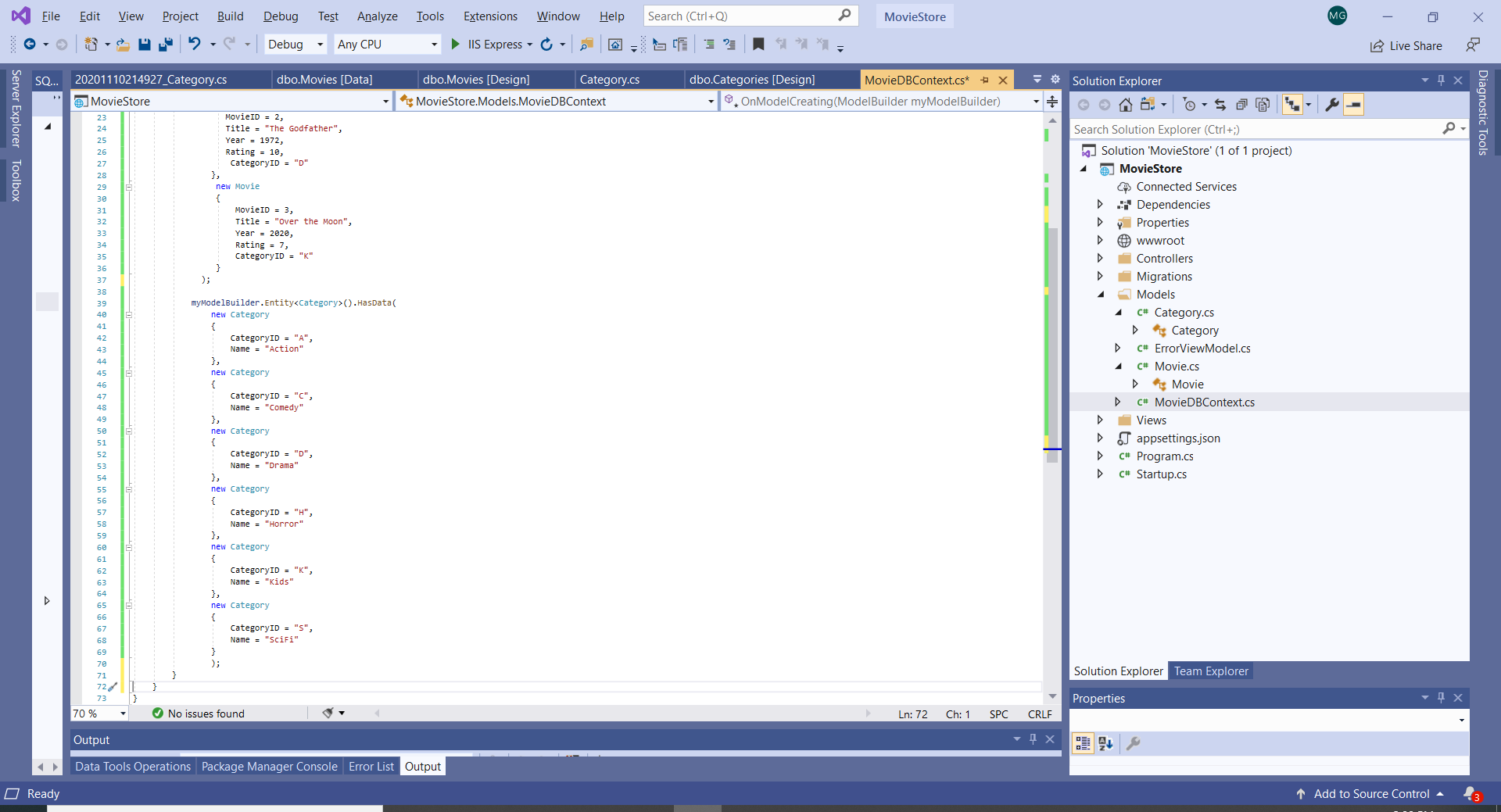
* 1. Update the MovieDBContext.

Before updating the DB, inspect the DB. In my case, I deleted previously the movie with ID=1. Therefore, I will not include it in the MovieDBContext.

The MovieDBContext class seeds the data for the new entities.







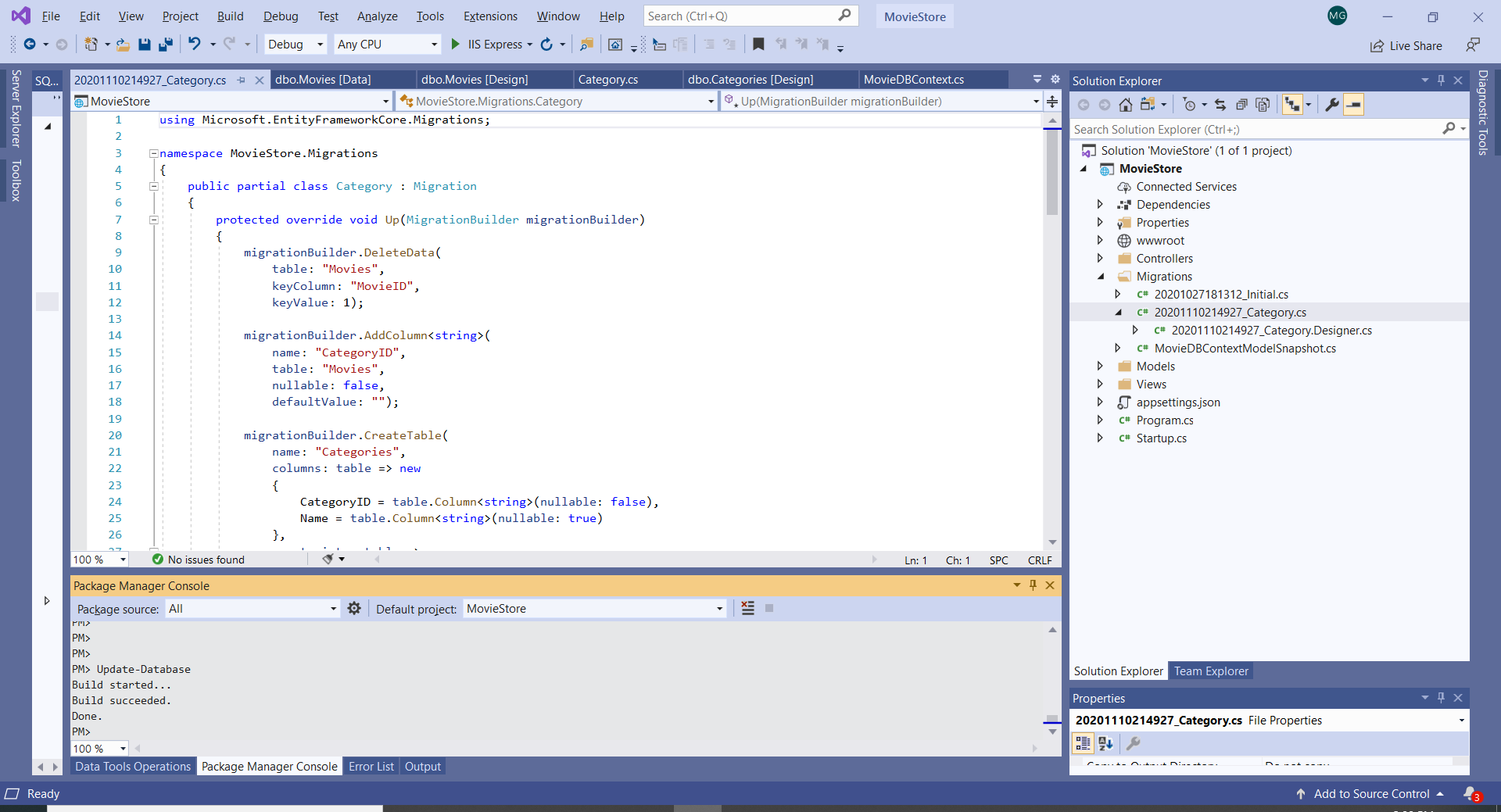
1. **Update the DB using Migrations**
2. Select Tools -> NuGet Package Manager -> Package Manager Console Command
3. In the PM console give the commands:
   * 1. “Add-Migration Category” and press Enter
     2. “Update Database” and press Enter



If successful, the DB is now updated.

The Update-Database executes the Up() method part of the EF.

1. Observe the Migration file that was created:



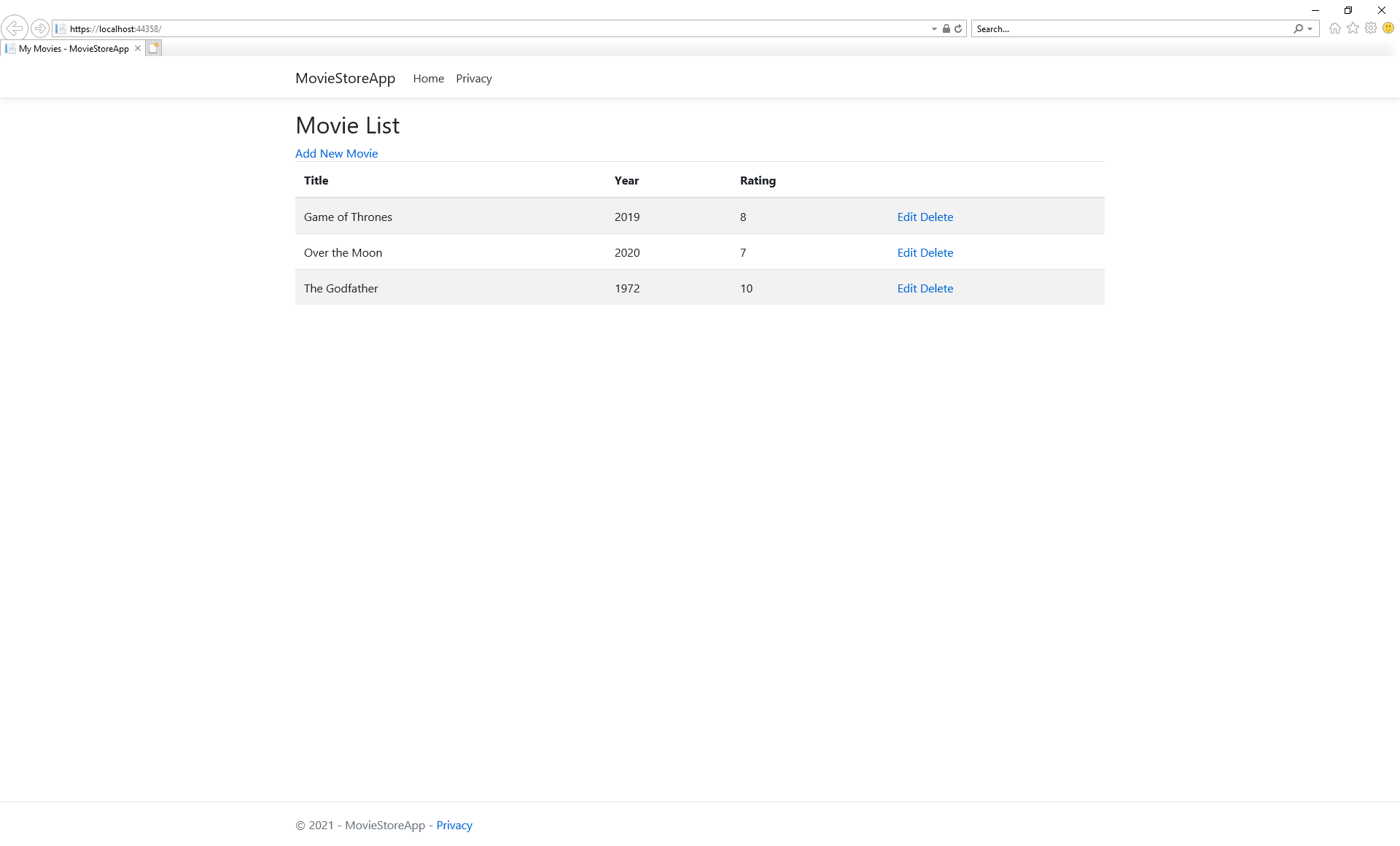
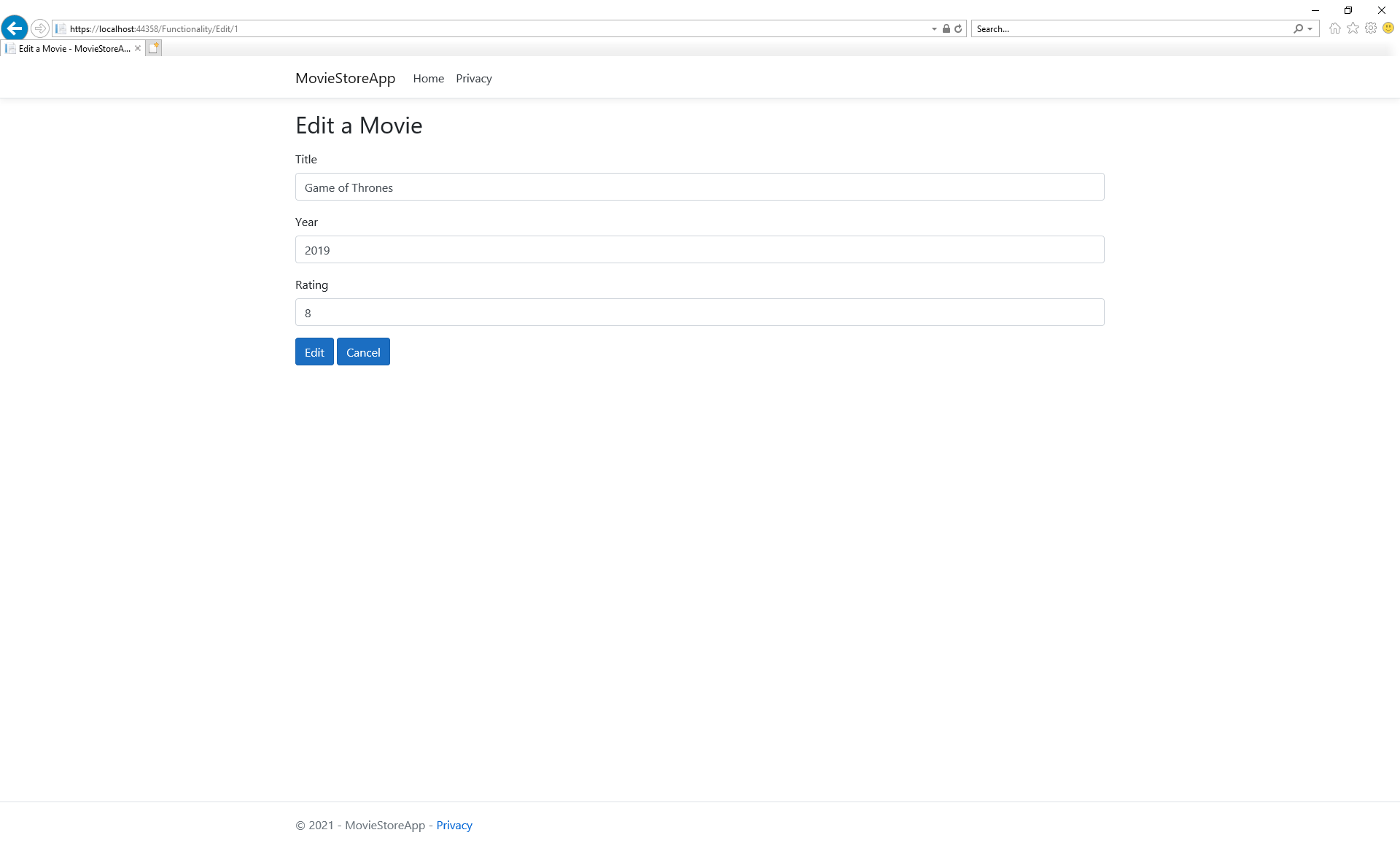
The new migration’s title is similar to this: 20201110214927\_Category.cs

Open the file and inspect it.

Observe that there are two methods: Up() and Down(). The Up() method creates the new table Category. The Down() method does the reverse operation - it removes changes and reverts the database to the state that is expected.

1. **Update the Controller: Select the Category data to display in the Movie List page**

Observe that category is not displayed in the movie table or included in the Add, New Movie or the Edit functionality. Additionally, the Views need to be updated.

1. 
2. 

The Category needs to be included in the controller’s action.

In the Home Controller include:

using Microsoft.EntityFrameworkCore;

This allow the use of EntityFrameworkCore methods. The Include() method is specific to EntityFrameworkCore and allows the selection of the Category data related to the each movie.

The following lambda expressions specify the related EF entities.

m =>m.Category

m => m.Title

Important: the Include() and OrderBy() methods do not execute at the database. They build the query expression that the ToList() method executes at the database.

Update the Index() method to include the Category:

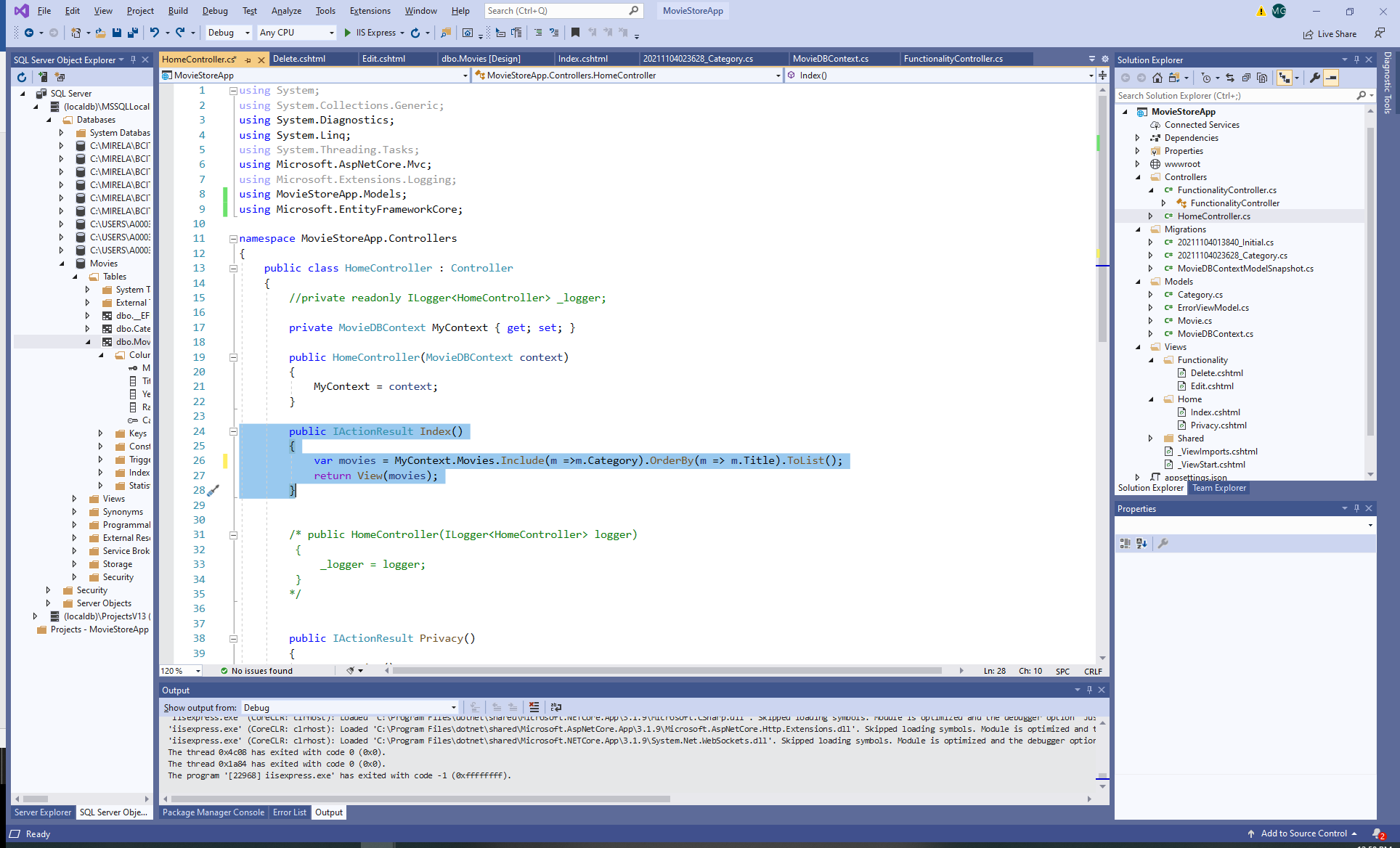
public IActionResult Index()

{

var movies = MyContext.Movies.Include(m =>m.Category).OrderBy(m => m.Title).ToList();

return View(movies);

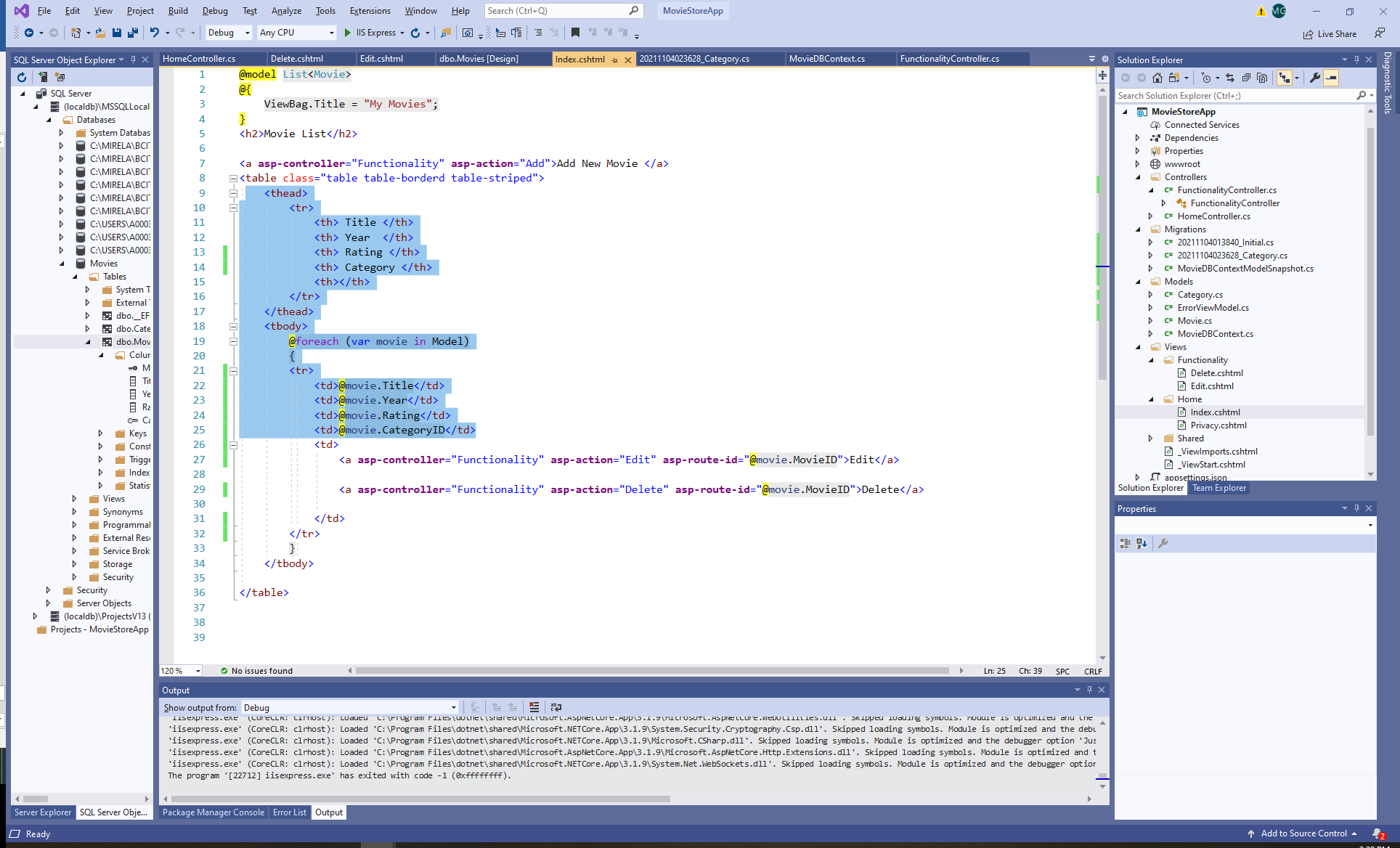
}



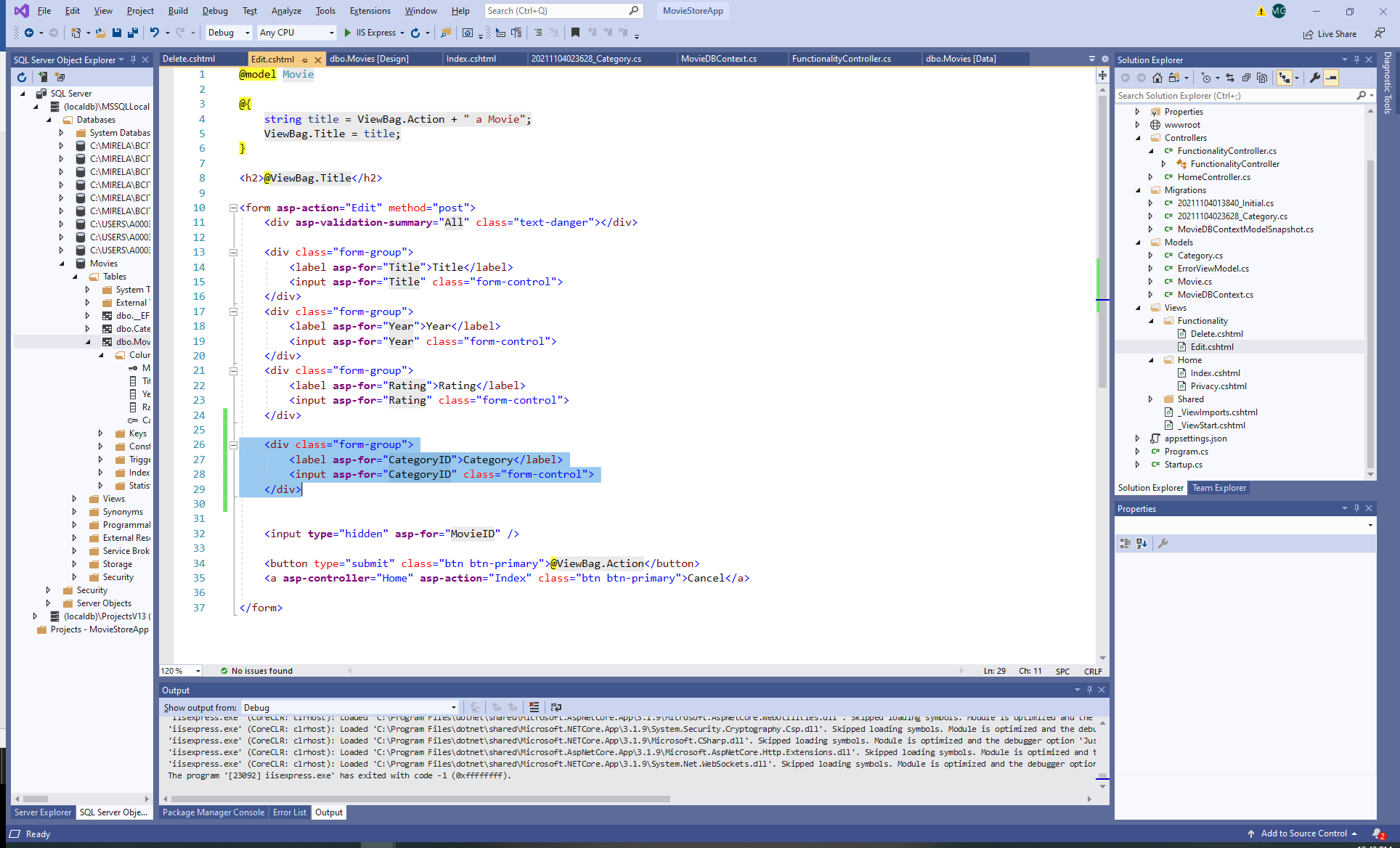
1. **Update the Views**

Views should be updated to display the Category.

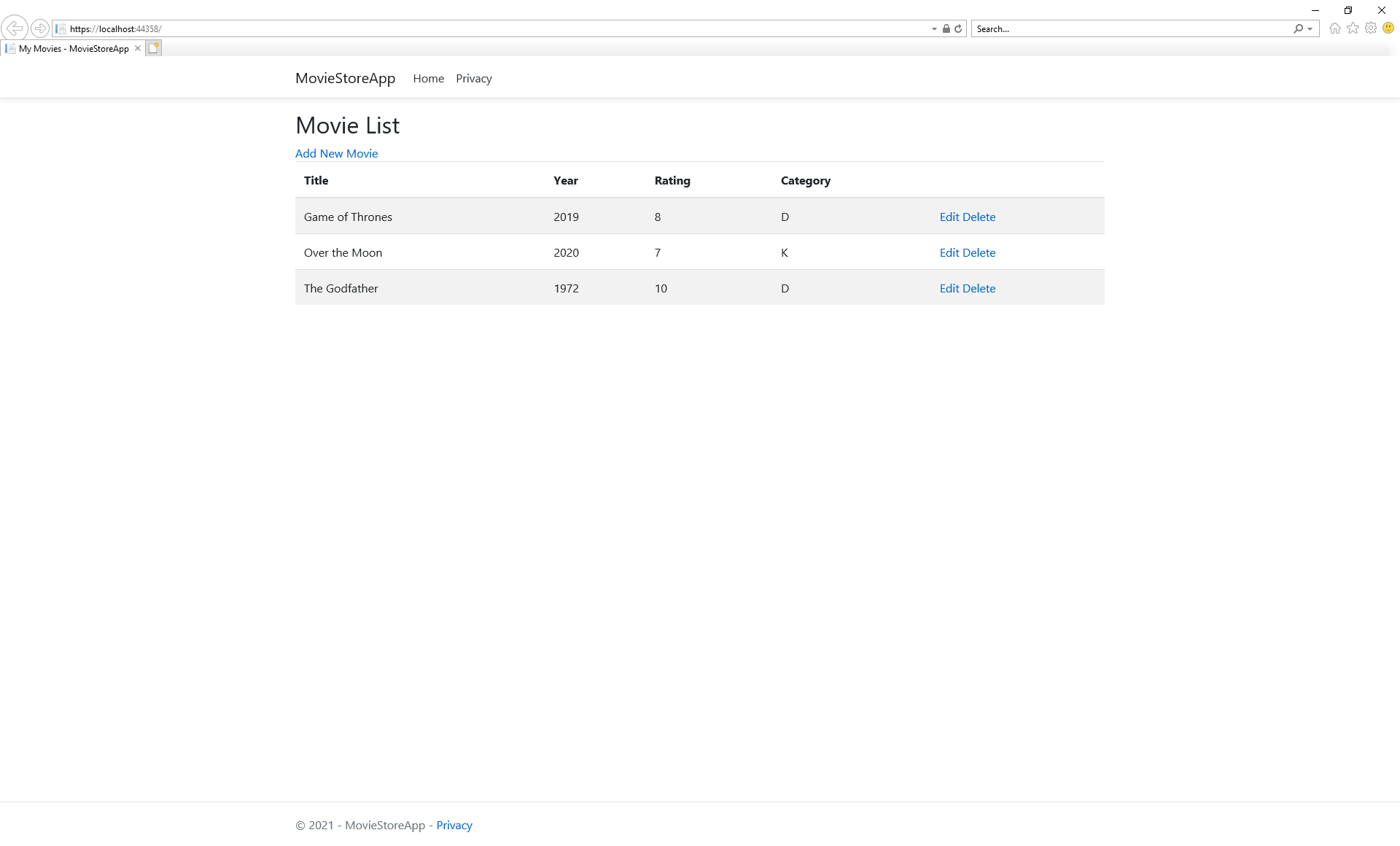
Home/Index View:



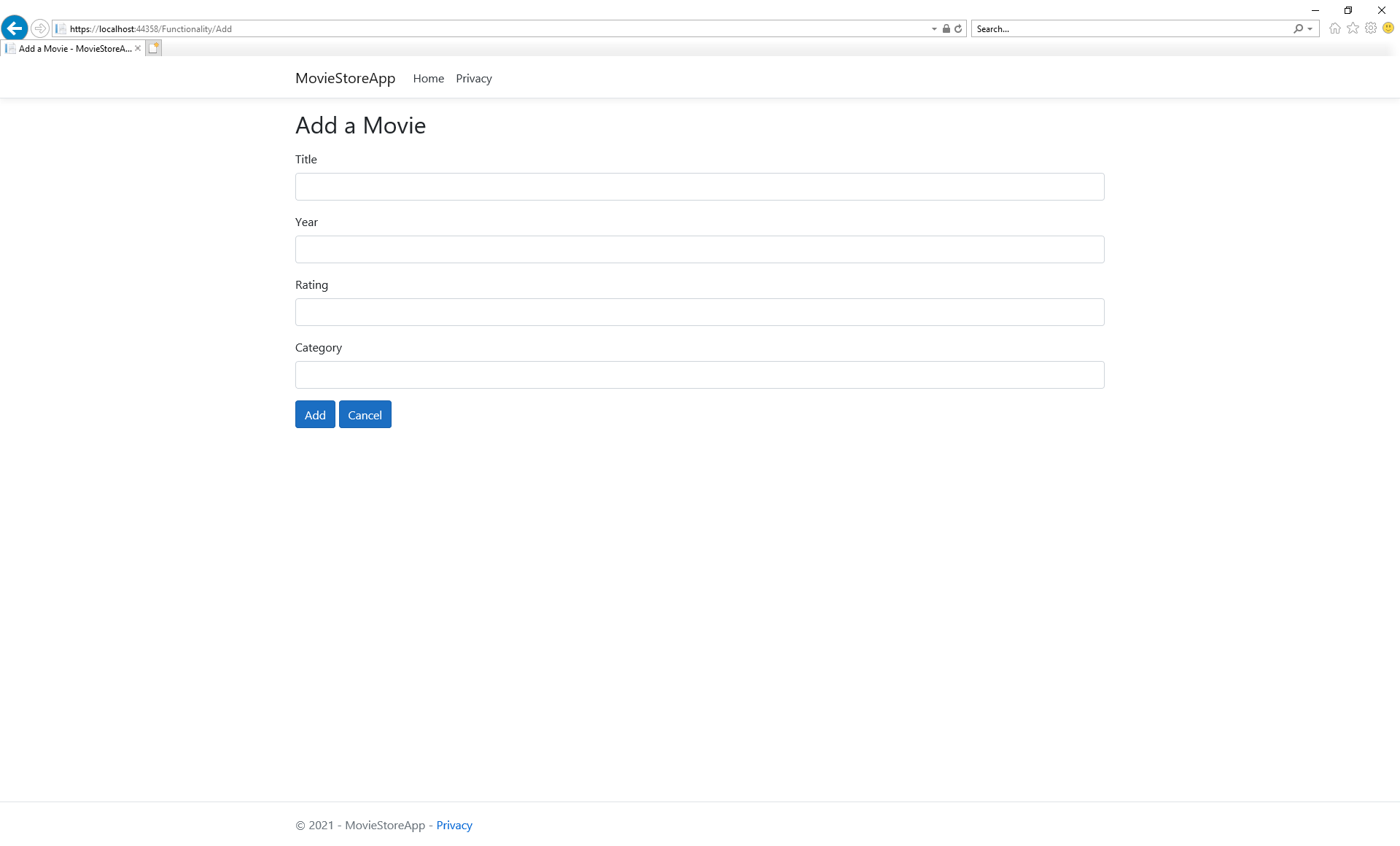
Functionality/Edit View:



Now the Category is displayed on the Movie List:



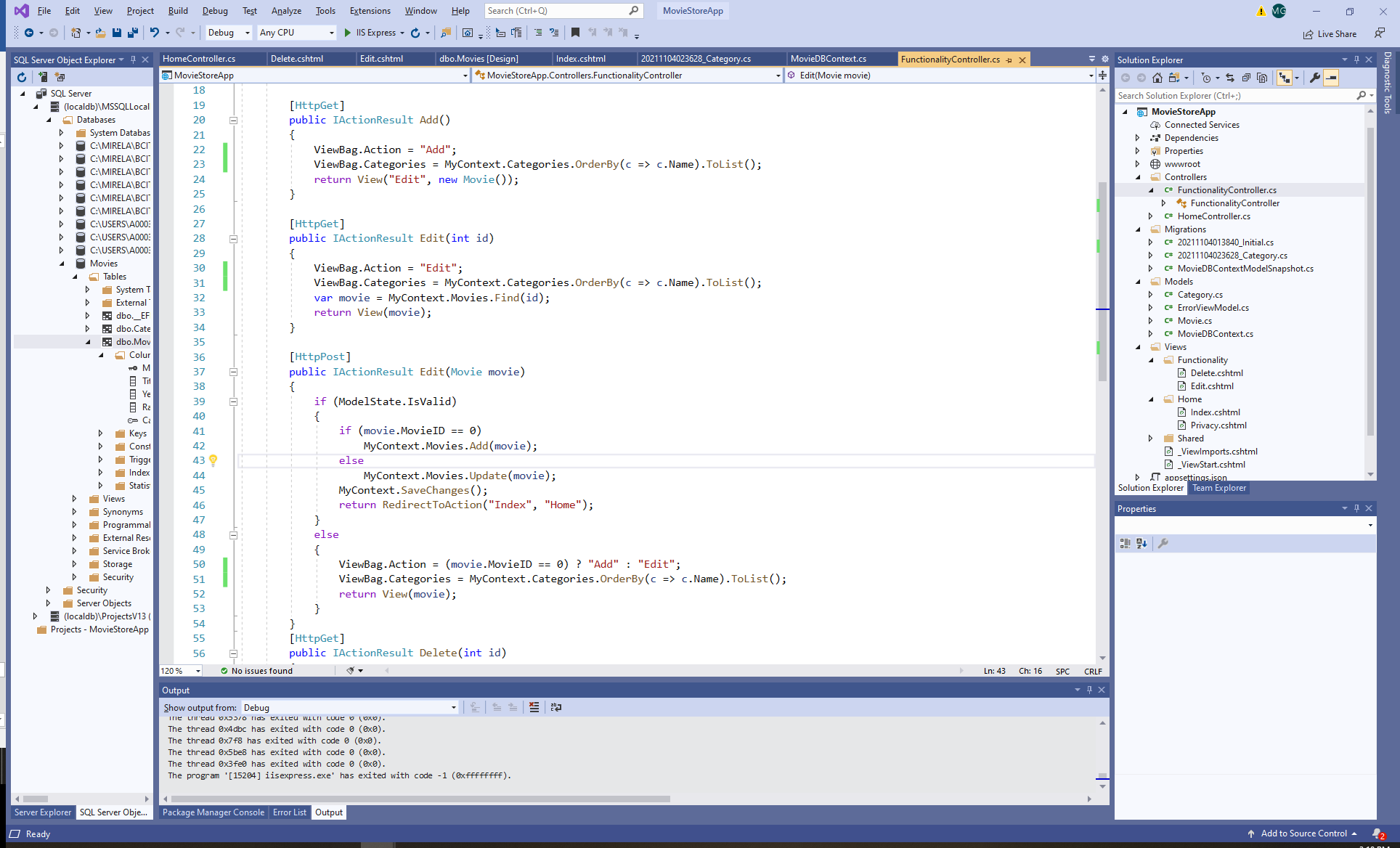




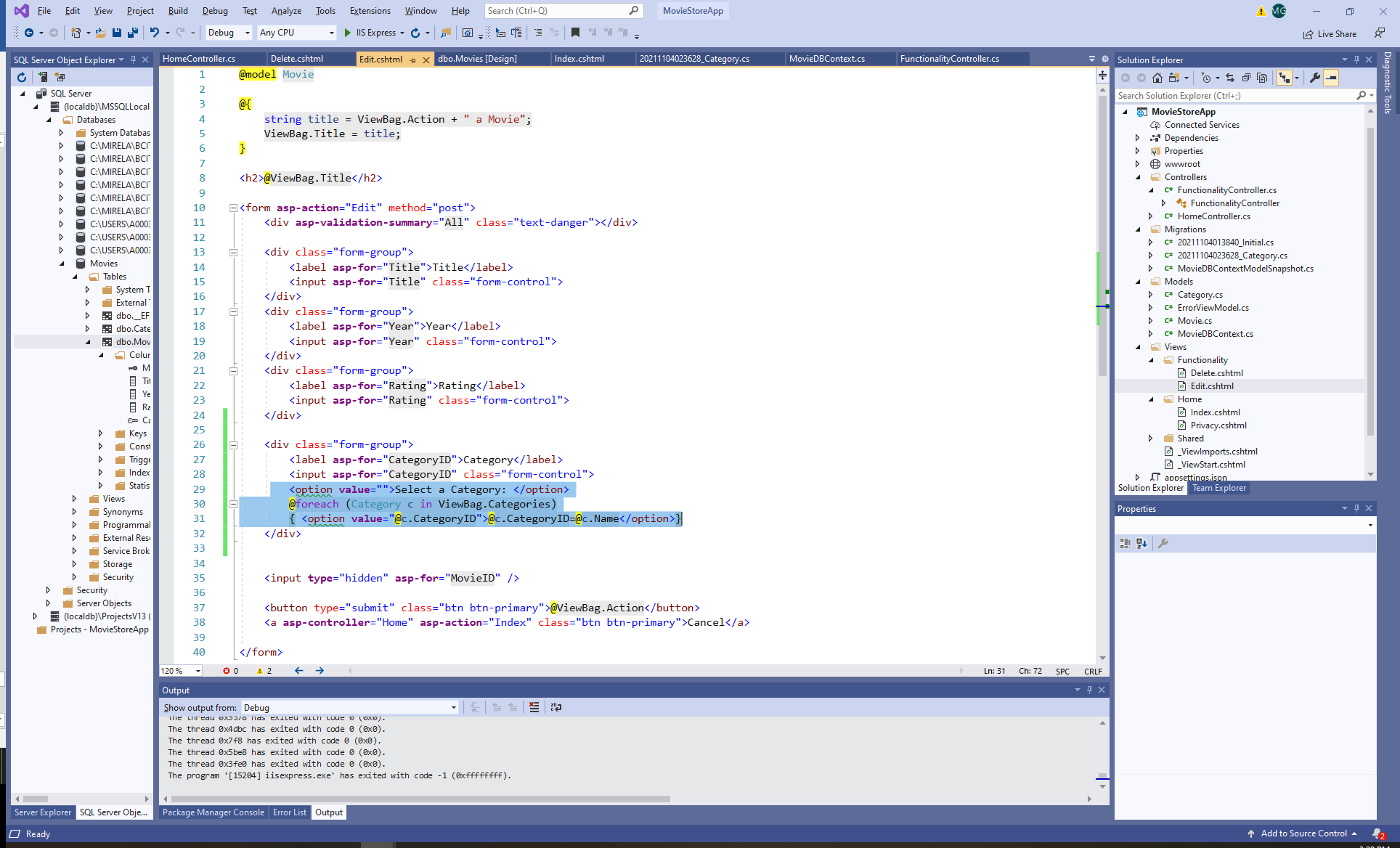
1. **Display Related Data**

To improve usability, display a list of Categories and their Category ID’s for the Add and Edit actions in the HttpGet and for Edit in the HttpPost. Include the following line of code:

ViewBag.Categories = MyContext.Categories.OrderBy(c => c.Name).ToList();



Update the Views accordingly:



Now the users can visualize the list of categories:

