Part 4, add a model to an ASP.NET Core MVC app

In this section, you add classes for managing movies in a database. These classes will be the "**M**odel" part of the **M**VC app.

You use these classes with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes you create are known as POCO classes (from **P**lain **O**ld **C**LR **O**bjects) because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial, you write the model classes first, and EF Core creates the database.

Add a data model class

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the *Models* folder > **Add** > **Class**. Name the file *Movie.cs*.

Update the *Movie.cs* file with the following code:

C#Copy
```csharp
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

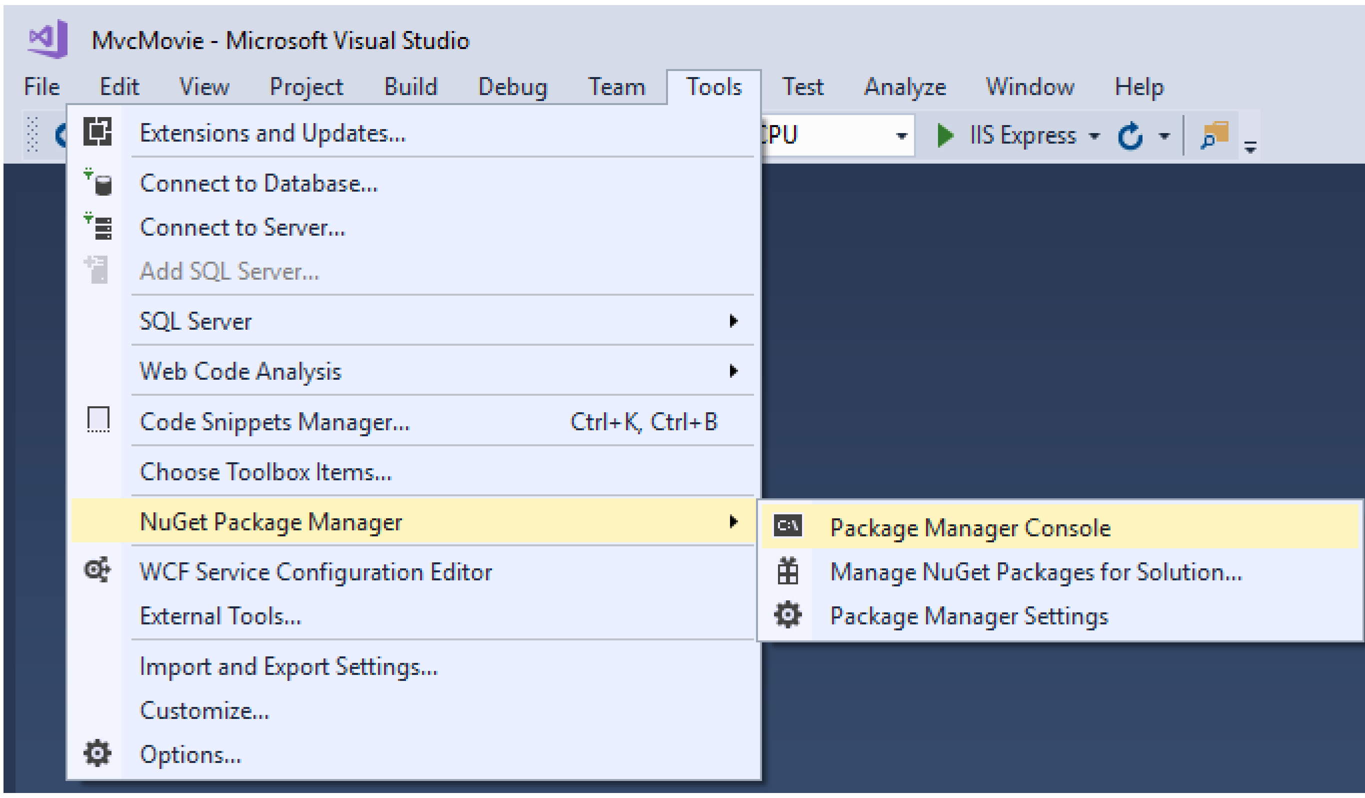
- The user is not required to enter time information in the date field.
- Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

Add NuGet packages

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



In the PMC, run the following command:

PowerShellCopy
```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The preceding command adds the EF Core SQL Server provider. The provider package installs the EF Core package as a dependency. Additional packages are installed automatically in the scaffolding step later in the tutorial.

Create a database context class

A database context class is needed to coordinate EF Core functionality (Create, Read, Update, Delete) for the `Movie` model. The database context is derived from [Microsoft.EntityFrameworkCore.DbContext]() and specifies the entities to include in the data model.

Create a *Data* folder.

Add a *Data/MvcMovieContext.cs* file with the following code:

C#Copy
```csharp
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}
```

The preceding code creates a [DbSet<Movie>]() property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

Register the database context

ASP.NET Core is built with [dependency injection (DI)](). Services (such as the EF Core DB context) must be registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial. In this section, you register the database context with the DI container.

Add the following `using` statements at the top of *Startup.cs*:

C#Copy
```csharp
using MvcMovie.Data;
using Microsoft.EntityFrameworkCore;
```

Add the following highlighted code in `Startup.ConfigureServices`:

- [Visual Studio]()

- Visual Studio Code / Visual Studio for Mac

C#Copy
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add a database connection string

Add a connection string to the *appsettings.json* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

JSONCopy
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

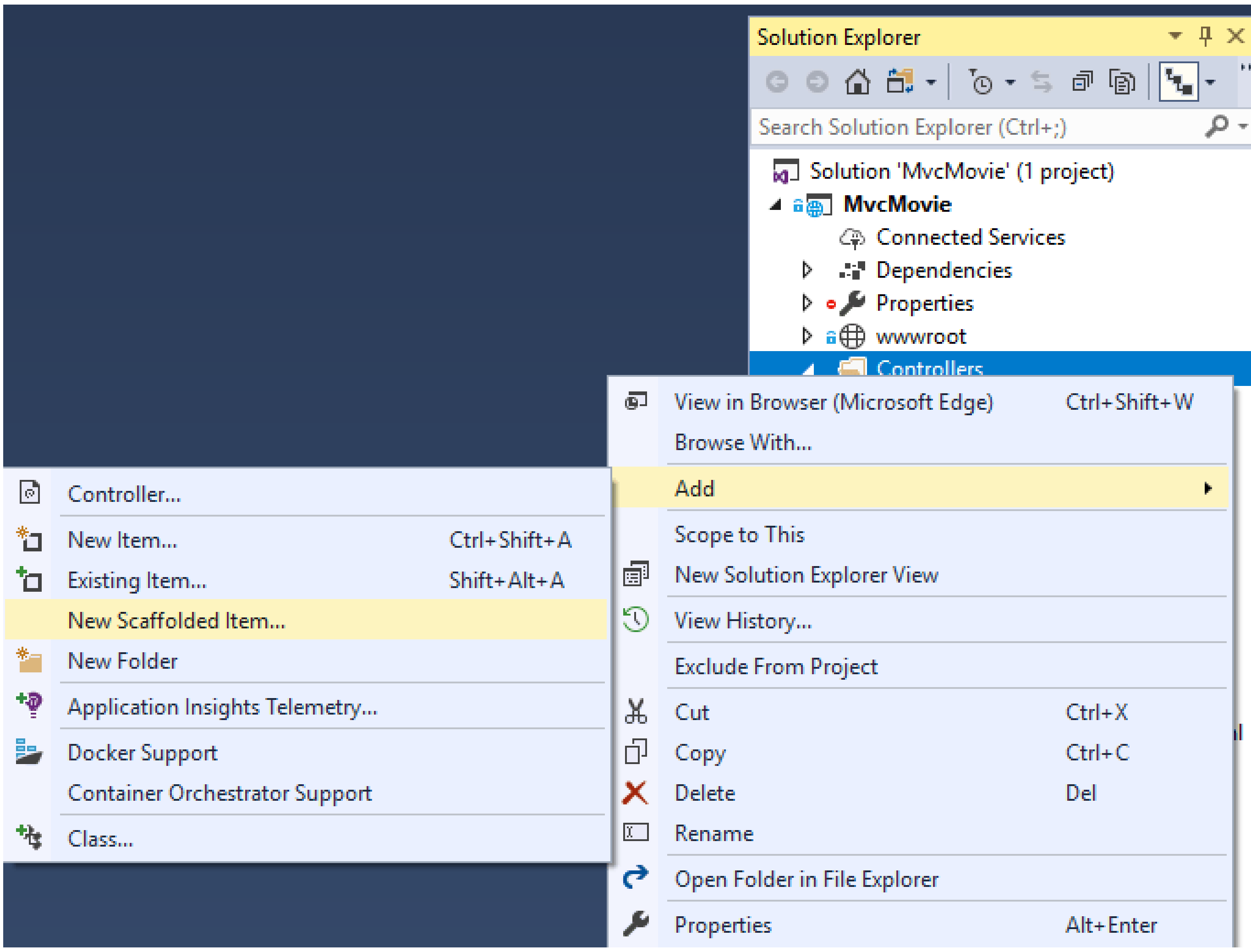
Build the project as a check for compiler errors.

Scaffold movie pages

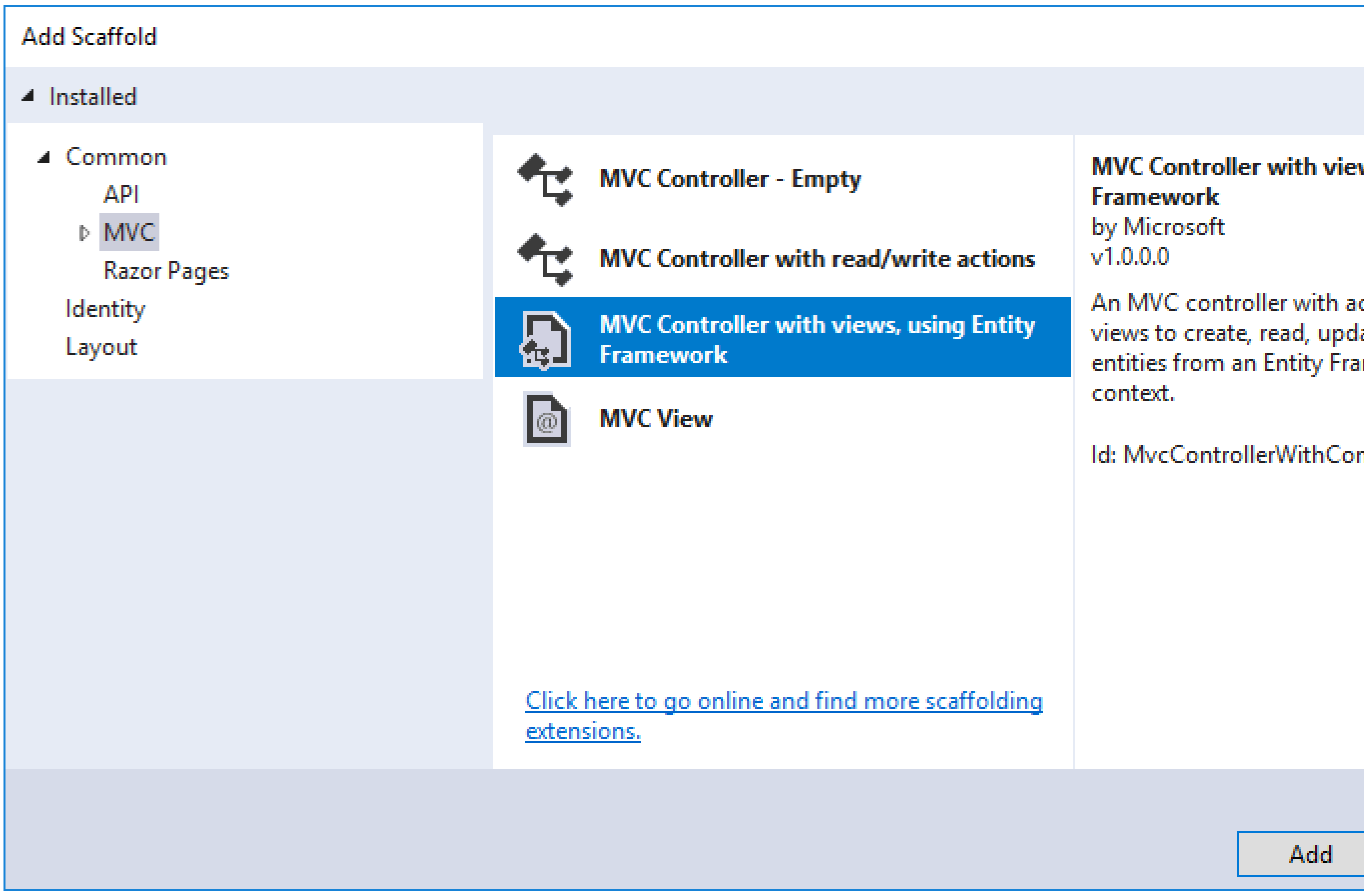
Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In **Solution Explorer**, right-click the *Controllers* folder **> Add > New Scaffolded Item**.



In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework > Add**.


Add Scaffold

Installed

Common

API

MVC

Razor Pages

Identity

Layout

MVC Controller - Empty

MVC Controller with read/write actions

MVC Controller with views, using Entity Framework

MVC View

Click here to go online and find more scaffolding extensions.

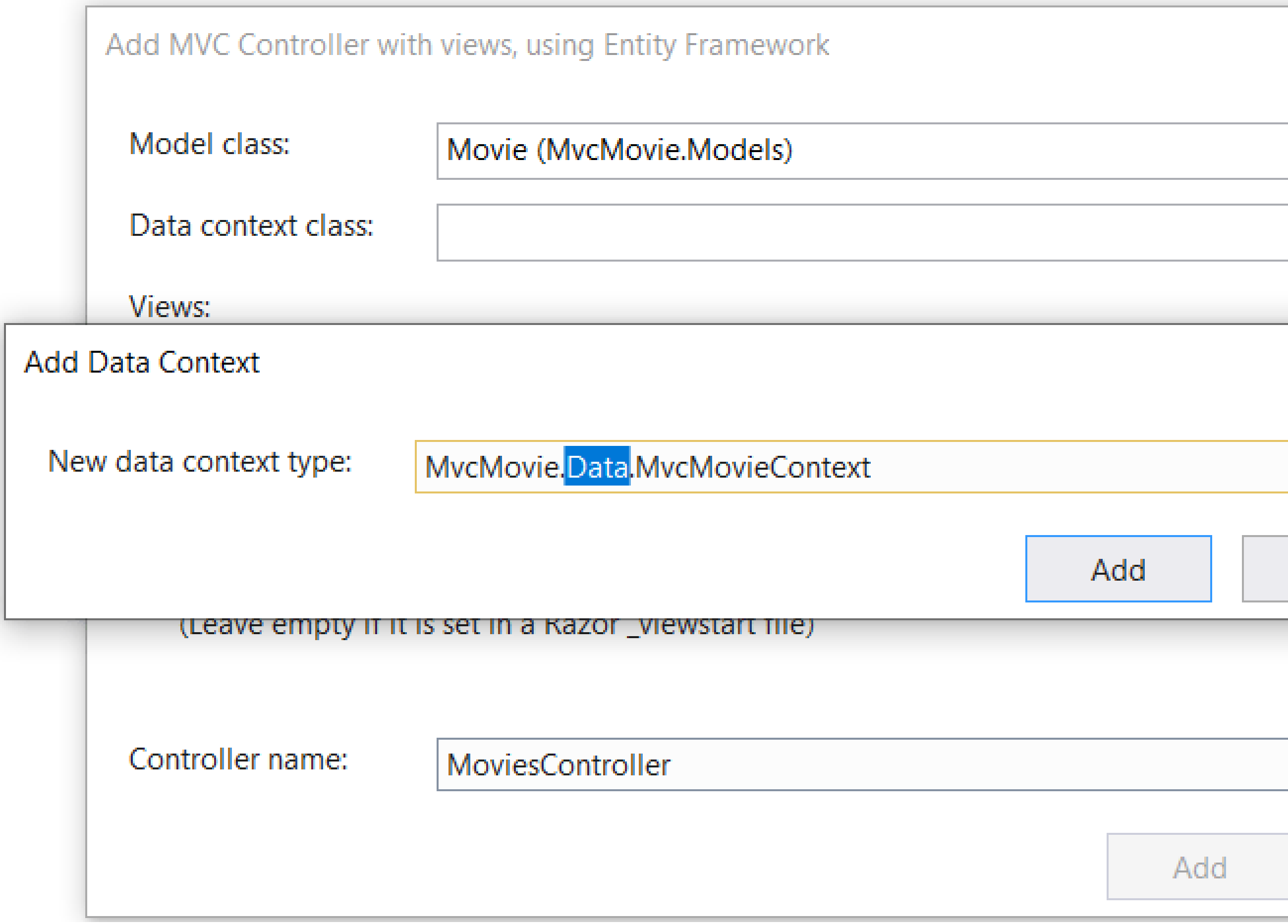
MVC Controller with vie
Framework
by Microsoft
v1.0.0.0

An MVC controller with ac
views to create, read, upda
entities from an Entity Fra
context.

Id: MvcControllerWithCor

Complete the **Add Controller** dialog:

- **Model class:** *Movie (MvcMovie.Models)*
- **Data context class:** *MvcMovieContext (MvcMovie.Data)*


Add MVC Controller with views, using Entity Framework

Model class: Movie (MvcMovie.Models)

Data context class:

Views:

Add Data Context

New data context type: MvcMovie.Data.MvcMovieContext

Add

(Leave empty if it is set in a Razor _viewstart file)

Controller name: MoviesController

Add

- **Views:** Keep the default of each option checked
- **Controller name:** Keep the default *MoviesController*
- Select **Add**

Visual Studio creates:

- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of these files is known as *scaffolding*.

You can't use the scaffolded pages yet because the database doesn't exist. If you run the app and click on the **Movie App** link, you get a *Cannot open database* or *no such table: Movie* error message.

Initial migration

Use the EF Core Migrations feature to create the database. Migrations is a set of tools that let you create and update a database to match your data model.

- Visual Studio

- Visual Studio Code / Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).

In the PMC, enter the following commands:

PowerShellCopy
```
Add-Migration InitialCreate
Update-Database
```

- `Add-Migration InitialCreate`: Generates a *Migrations/{timestamp}_InitialCreate.cs* migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.
- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the *Migrations/{time-stamp}_InitialCreate.cs* file, which creates the database.

  The database update command generates the following warning:

  No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

  You can ignore that warning, it will be fixed in a later tutorial.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

The InitialCreate class

Examine the *Migrations/{timestamp}_InitialCreate.cs* migration file:

C#Copy
```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Movie",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                                SqlServerValueGenerationStrategy.IdentityColumn),
                Title = table.Column<string>(nullable: true),
                ReleaseDate = table.Column<DateTime>(nullable: false),
                Genre = table.Column<string>(nullable: true),
                Price = table.Column<decimal>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Movie", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Movie");
    }
}
```

The `Up` method creates the Movie table and configures `Id` as the primary key. The `Down` method reverts the schema changes made by the `Up` migration.

Test the app

- Run the app and click the **Movie App** link.

  If you get an exception similar to one of the following:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

ConsoleCopy
```
SqlException: Cannot open database "MvcMovieContext-1" requested by the login. The
login failed.
```

You probably missed the migrations step.

- Test the **Create** page. Enter and submit data.

  Note

  You may not be able to enter decimal commas in the `Price` field. To support **jQuery validation** for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see **this GitHub issue**.

- Test the **Edit**, **Details**, and **Delete** pages.

Dependency injection in the controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Open the *Controllers/MoviesController.cs* file and examine the constructor:
C#Copy
```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

C#Copy
```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).

- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

```
https://localhost:5001/movies/details?id=1
```

The `id` parameter is defined as a nullable type (`int?`) in case an ID value isn't provided.

A lambda expression is passed in to `FirstOrDefaultAsync` to select movie entities that match the route data or query string value.

C#Copy
```csharp
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

C#Copy
```csharp
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

CSHTMLCopy
```cshtml
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

CSHTMLCopy
```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

C#Copy
```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When the movies controller was created, scaffolding included the following `@model` statement at the top of the *Index.cshtml* file:

CSHTMLCopy
```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

CSHTMLCopy
```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}
```

```
<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```