Lab: ASP.NET Core MVC Microservice

In this lab

1. Prerequisites
2. Create a web app

This lab teaches ASP.NET Core MVC web development with controllers and views.

At the end of the lab, you'll have an app that manages and displays movie data. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

Create a web app

- Open the integrated terminal.
- Change to the directory (`cd`) that will contain the project.
- Run the following command:

```
```
dotnet new mvc -o MvcMovie

cd MvcMovie/
```


-   `dotnet new mvc -o MvcMovie`: Creates a new ASP.NET Core MVC
    project in the *MvcMovie* folder.
```

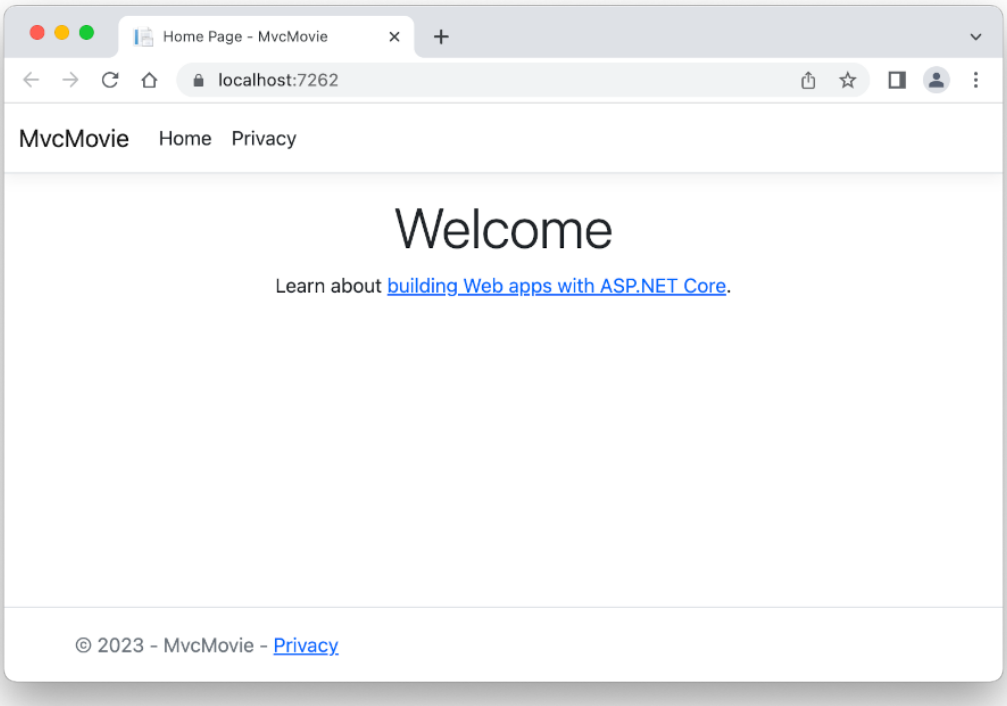
Run the app

Build and run the app by executing the following command in your terminal:

`dotnet run`

Wait for the app to display that it's listening and then open a browser and navigate to `https://PORT-YOUR_GITPOD_URL.gitpod.io`

The following image shows the app:



Part 2, add a controller to an ASP.NET Core MVC app

In this lab

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this lab, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

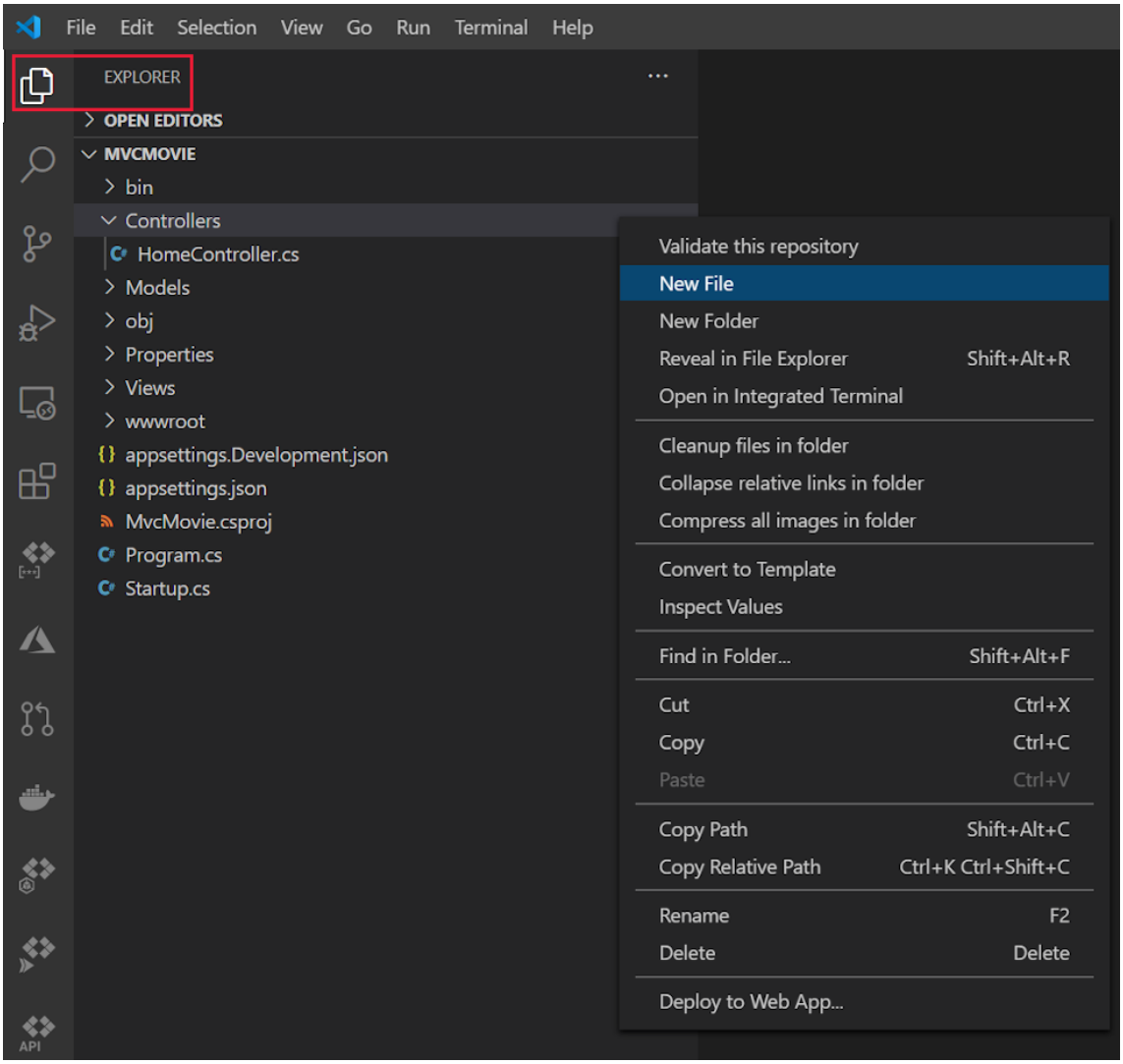
- `https://PORT-YOUR_GITPOD_URL.gitpod.io/Home/Privacy` : specifies the `Home` controller and the `Privacy` action.
- `https://PORT-YOUR_GITPOD_URL.gitpod.io/Movies/Edit/5` : is a request to edit the movie with ID=5 using the `Movies` controller and the `Edit` action, which are detailed later in the tutorial.

The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this lab series while building a movie app. The MVC project contains folders for the *Controllers* and *Views*.

Add a controller

Select the **EXPLORER** icon and then control-click (right-click) **Controllers > New File** and name the new file `HelloWorldController.cs` .



Replace the contents of `Controllers/HelloWorldController.cs` with the following code:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

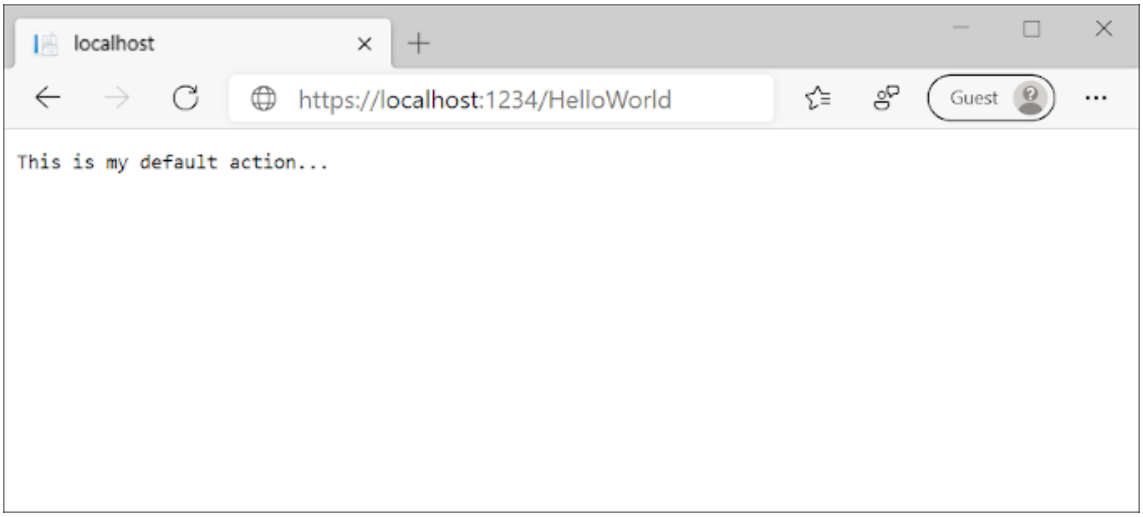
- Is a targetable URL in the web application, such as `https://PORT-YOUR_GITPOD_URL.gitpod.io/HelloWorld`.
- Combines:
  - The protocol used: `HTTPS`.
  - The network location of the web server, including the TCP port: `PORT-YOUR_GITPOD_URL.gitpod.io`.
  - The target URI: `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL.

The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial, the scaffolding engine is used to generate `HTTP POST` methods, which update data.

Run the app without the debugger.

Append `/HelloWorld` to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default [URL routing logic] used by MVC, uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

The routing format is set in the `Program.cs` file.

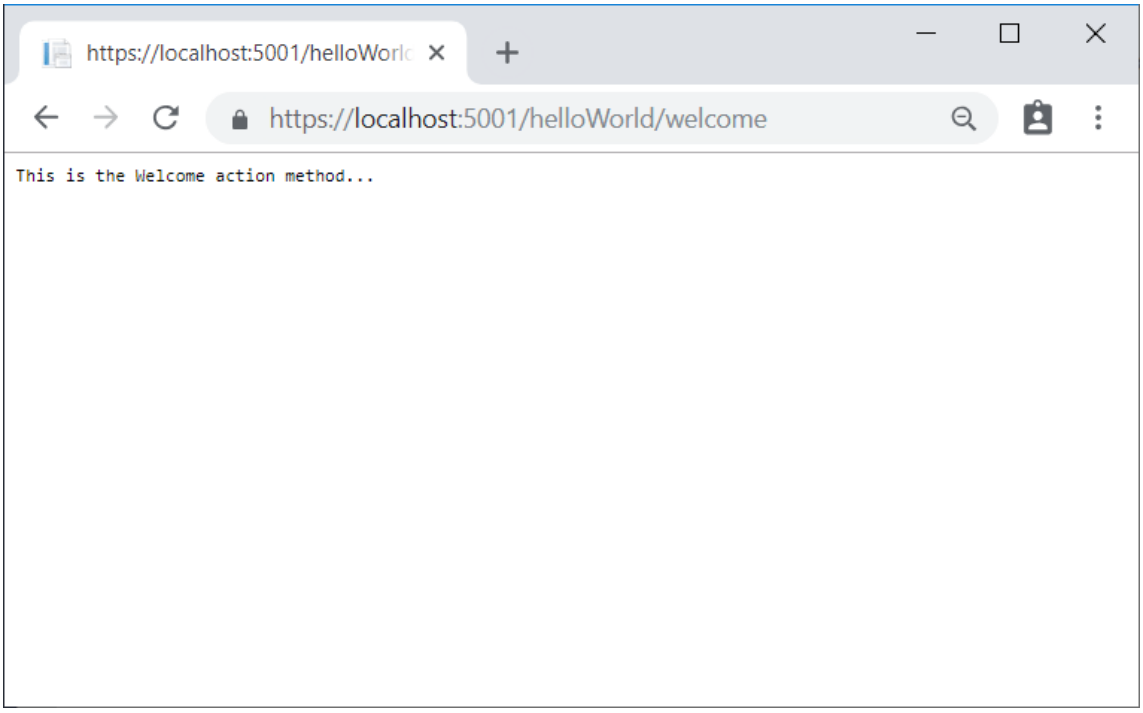
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So `PORT-YOUR_GITPOD_URL.gitpod.io/HelloWorld` maps to the **HelloWorld** Controller class.
- The second part of the URL segment determines the action method on the class. So `PORT-YOUR_GITPOD_URL.gitpod.io/HelloWorld/Index` causes the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `PORT-YOUR_GITPOD_URL.gitpod.io/HelloWorld` and the `Index` method was called by default. `Index` is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to: `https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld/Welcome`. Replace `{PORT}` with your port number.

The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`.

Change the `Welcome` method to include two parameters as shown in the following code.

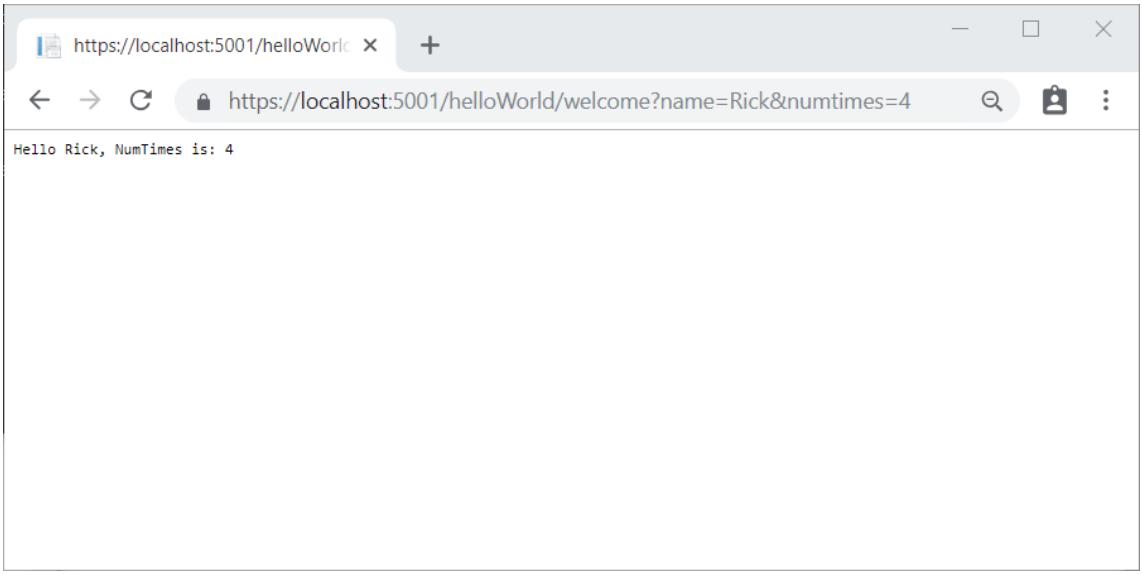
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input, such as through JavaScript.
- Uses [Interpolated Strings] in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to: `https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld/Welcome?name=Rick&numtimes=4`. Replace `{PORT}` with your port number.

Try different values for `name` and `numtimes` in the URL. The MVC [model binding] system automatically maps the named parameters from the query string to parameters in the method. See [Model Binding] for more information.



In the previous image:

- The URL segment `Parameters` isn't used.
- The `name` and `numTimes` parameters are passed in the query string.
- The `?` (question mark) in the above URL is a separator, and the query string follows.
- The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld/Welcome/3?name=Rick`

In the preceding URL:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` starts the query string.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

In the preceding example:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` (in `id?`) indicates the `id` parameter is optional.

Part 3, add a view to an ASP.NET Core MVC app

In this lab

1. Add a view
2. Change views and layout pages
3. Change the title, footer, and menu link in the layout file
4. Passing Data from the Controller to the View

In this section, you modify the `HelloWorldController` class to use [Razor] view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

- Have a `.cshtml` file extension.
- Provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

```
public IActionResult Index()
{
    return View();
}
```

The preceding code:

- Calls the controller's [View] method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as *action methods*. For example, the `Index` action method in the preceding code.
- Generally return an [IActionResult] or a class derived from [ActionResult], not a type like `string`.

Add a view

Add an `Index` view for the `HelloWorldController`:

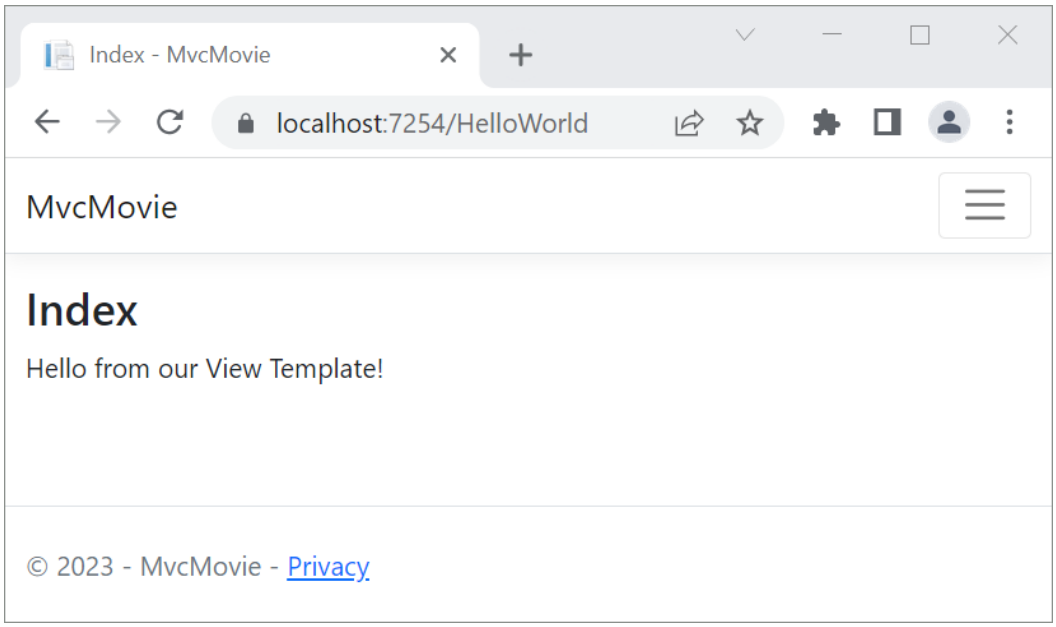
- Add a new folder named *Views/HelloWorld*.
- Add a new file to the *Views/HelloWorld* folder, and name it `Index.cshtml`.

Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

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

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld`:

- The `Index` method in the `HelloWorldController` ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, `Index` in this example. The view template `/Views/HelloWorld/Index.cshtml` is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view:



Change views and layout pages

Select the menu links **MvcMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Views/Shared/_Layout.cshtml` file.

Open the `Views/Shared/_Layout.cshtml` file.

[Layout] templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the *`Views/Home/Privacy.cshtml`* view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the `Views/Shared/_Layout.cshtml` file with the following markup:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white
border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies" asp-
action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-
between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2023 - Movie App - <a asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

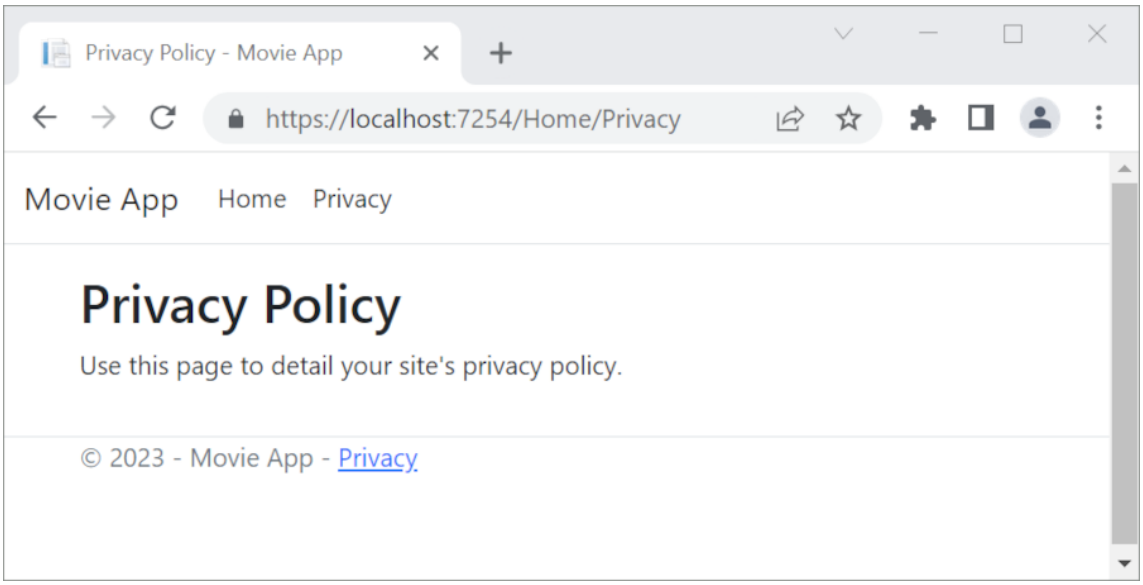
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` [anchor Tag Helper attribute] and attribute value was omitted because this app isn't using [Areas].

Note: The `Movies` controller hasn't been implemented. At this point, the `Movie App` link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**



Select the **Home** link.

Notice that the title and anchor text display **Movie App**. The changes were made once in the layout template and all pages on the site reflect the new link text and new title.

Examine the `Views/_ViewStart.cshtml` file:

```
@{
    Layout = "_Layout";
}
```

The `Views/_ViewStart.cshtml` file brings in the `Views/Shared/_Layout.cshtml` file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Open the `Views/HelloWorld/Index.cshtml` view file.

Change the title and `<h2>` element as highlighted in the following:

```
@{
    ViewData["Title"] = "Movie List";
```

```
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so it's clear which part of the code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

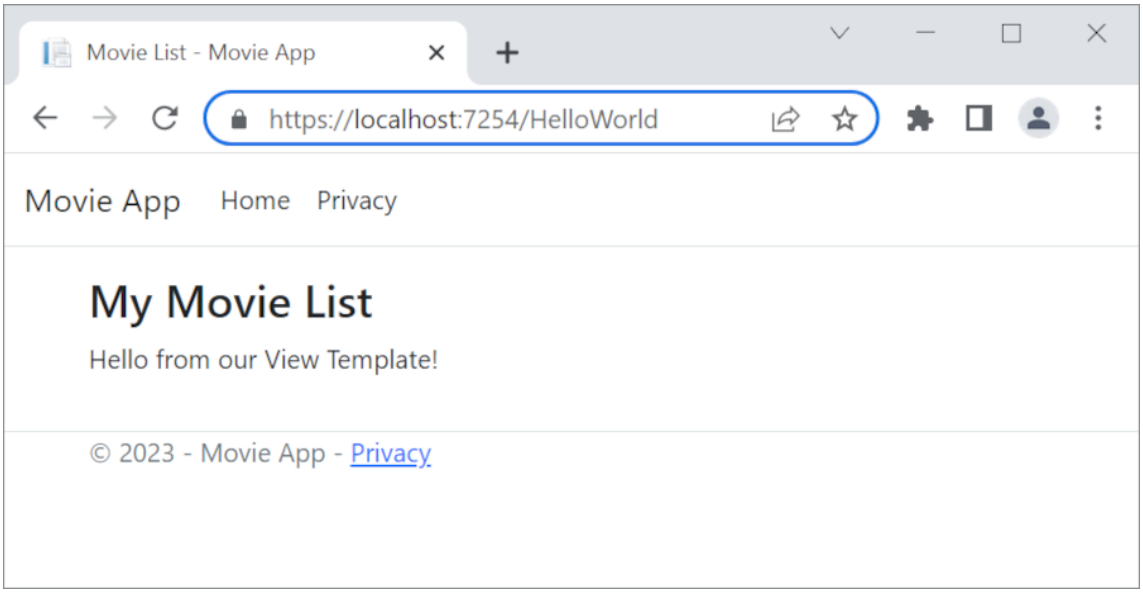
Save the change and navigate to `https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld`.

Notice that the following have changed:

- Browser title.
- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

The content in the `Index.cshtml` view template is merged with the `Views/Shared/_Layout.cshtml` view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see [Layout].



The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

View templates should **not**:

- Do business logic
- Interact with a database directly.

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and an `ID` parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary. The view template can then access the dynamic data.

In `HelloWorldController.cs`, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary.

The `ViewData` dictionary is a dynamic object, which means any type can be used. The `ViewData` object has no defined properties until something is added. The [MVC model binding system] automatically maps the named parameters `name` and `numTimes` from the query string to parameters in the method. The complete `HelloWorldController`:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;
        return View();
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named `Views/HelloWorld/Welcome.cshtml`.

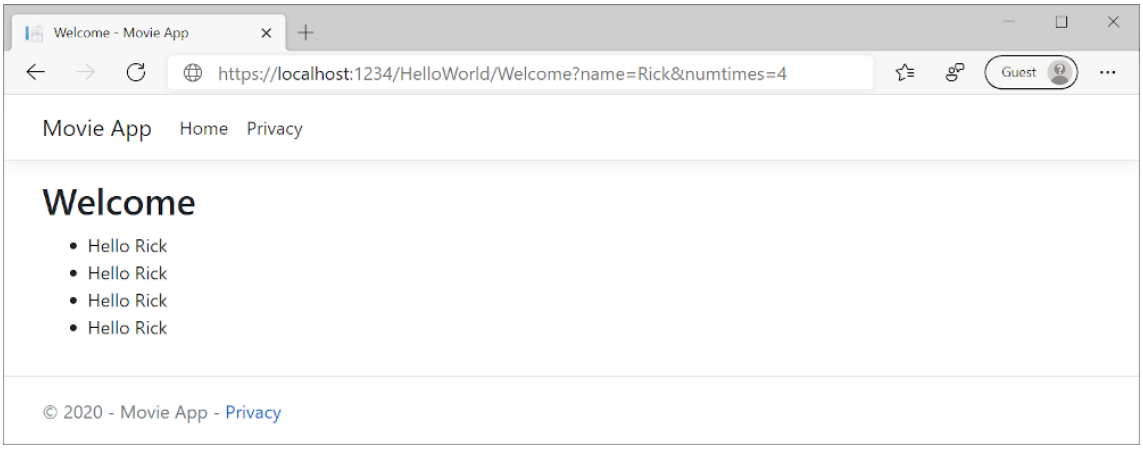
You'll create a loop in the `Welcome.cshtml` view template that displays "Hello" `NumTimes`. Replace the contents of `Views/HelloWorld/Welcome.cshtml` with the following:

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

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://{PORT}-YOUR_GITPOD_URL.gitpod.io/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the [MVC model binder]. The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the preceding sample, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the `ViewData` dictionary approach.

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

In this lab

1. Add a data model class
2. Add NuGet packages
3. Scaffold movie pages

4. Initial migration
5. Test the app
6. Dependency injection in the controller
7. Strongly typed models and the @model directive

In this lab, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with [Entity Framework 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 created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this lab, model classes are created first, and EF Core creates the database. Add a data model class

---

Add a file named `Movie.cs` to the *Models* folder.

Update the `Models/Movie.cs` file with the following code:

```
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 isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

The question mark after `string` indicates that the property is nullable.

Add NuGet packages

Run the following .NET CLI commands:

```
dotnet tool uninstall --global dotnet-aspnet-codegenerator
dotnet tool install --global dotnet-aspnet-codegenerator
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SQLite
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Build the project as a check for compiler errors:

`dotnet build`

Scaffold movie pages

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

Open a command window in the project directory. The project directory is the directory that contains the `Program.cs` and `.csproj` files.

Run the following command:

```
dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc
MvcMovie.Data.MvcMovieContext --relativeFolderPath Controllers --useDefaultLayout --
referenceScriptLibraries --databaseProvider sqlite
```

Scaffolding creates the following:

- A movies controller: `Controllers/MoviesController.cs`
- Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: `Views/Movies/*.cshtml`
- A database context class: `Data/MvcMovieContext.cs`

Scaffolding updates the following:

- Registers the database context in the `Program.cs` file
- Adds a database connection string to the `appsettings.json` file.

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

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no such table: Movie* error message.

Build the app to verify that there are no errors.

Use SQLite for development, SQL Server for production

The following highlighted code in `Program.cs` shows how to use SQLite in development and SQL Server in production.

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("MvcMovieContext")));
}
else
{
    builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("ProductionMvcMovieContex

}
```

Initial migration

Use the EF Core [Migrations] feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

Run the following .NET CLI commands:

```
dotnet ef migrations add InitialCreate
```

```
dotnet ef database update
```

- `ef migrations add 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. This is the first migration, so the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class, in the `Data/MvcMovieContext.cs` file.
- `ef database update` : 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.

Test the app

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the `dotnet ef database update` command in the [migrations step

```
SqliteException: SQLite Error 1: 'no such table: Movie'.
```

Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from [Microsoft.EntityFrameworkCore.DbContext] and specifies the entities to include in the data model.

Scaffolding creates the `Data/MvcMovieContext.cs` database context class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

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

```
        public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a [DbSet<Movie>] property that represents the movies in the database.

Examine the generated database connection string

Scaffolding added a connection string to the `appsettings.json` file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Data Source=MvcMovie.db"
  }
}
```

The `InitialCreate` class

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

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace MvcMovie.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Movie",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Title = table.Column<string>(type: "nvarchar(max)", nullable:
true),
                    ReleaseDate = table.Column<DateTime>(type: "datetime2", nullable:
false),
                    Genre = table.Column<string>(type: "nvarchar(max)", nullable:
true),
                    Price = table.Column<decimal>(type: "decimal(18,2)", nullable:
false)
```

```
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Movie", x => x.Id);
                });
        }

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

In the preceding code:

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

Dependency injection in the controller

Open the `Controllers/MoviesController.cs` file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

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

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

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

Strongly typed models and the `@model` directive

Earlier in this lab, 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 provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the `MoviesController` class and views.

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

```
// 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://PORT-YOUR_GITPOD_URL.gitpod.io/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.

The `id` can be passed in with a query string, as in the following example:

`https://PORT-YOUR_GITPOD_URL.gitpod.io/movies/details?id=1`

The `id` parameter is defined as a [nullable type] (`int?`) in cases when the `id` value isn't provided.

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

```
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:

```
return View(movie);
```

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

```
@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:

```
@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:

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

The code returns [problem details] if the `Movie` property of the data context is null.

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

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

The `@model` directive allows access to 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:

```
@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>
```

Because the `Model` object is strongly typed as an `IEnumerable<Movie>` object, each item in the loop is typed as `Movie`. Among other benefits, the compiler validates the types used in the code.