

Get started with ASP.NET Core MVC

Article • 11/08/2021 • 15 minutes to read •  +14

[Is this page helpful?](#)

In this article

[Prerequisites](#)

[Create a web app](#)

[Visual Studio help](#)

By [Rick Anderson](#)

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the [Razor Pages](#) version of this tutorial, which provides an easier starting point. See [Choose an ASP.NET Core UI](#), which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

At the end of the series, 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.

[View or download sample code](#) ([how to download](#)).

Prerequisites

[Visual Studio](#)

[Visual Studio Code](#)

[Visual Studio for Mac](#)

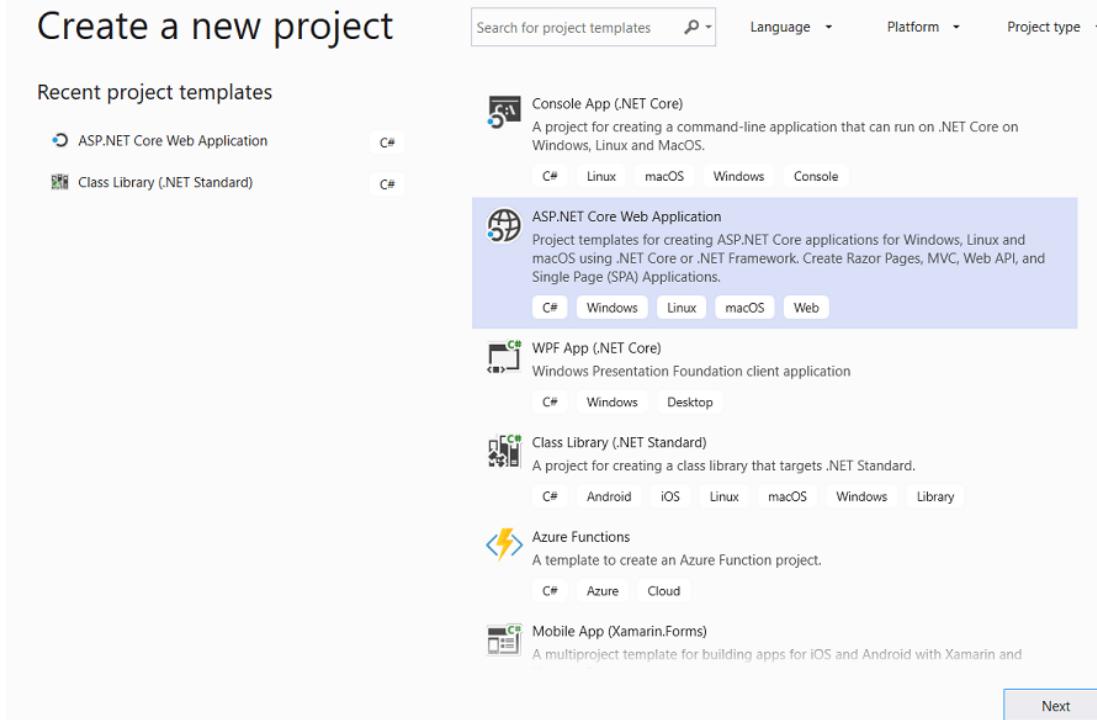
- [Visual Studio 2019 16.4 or later](#) with the **ASP.NET and web development workload**

- .NET Core 3.1 SDK

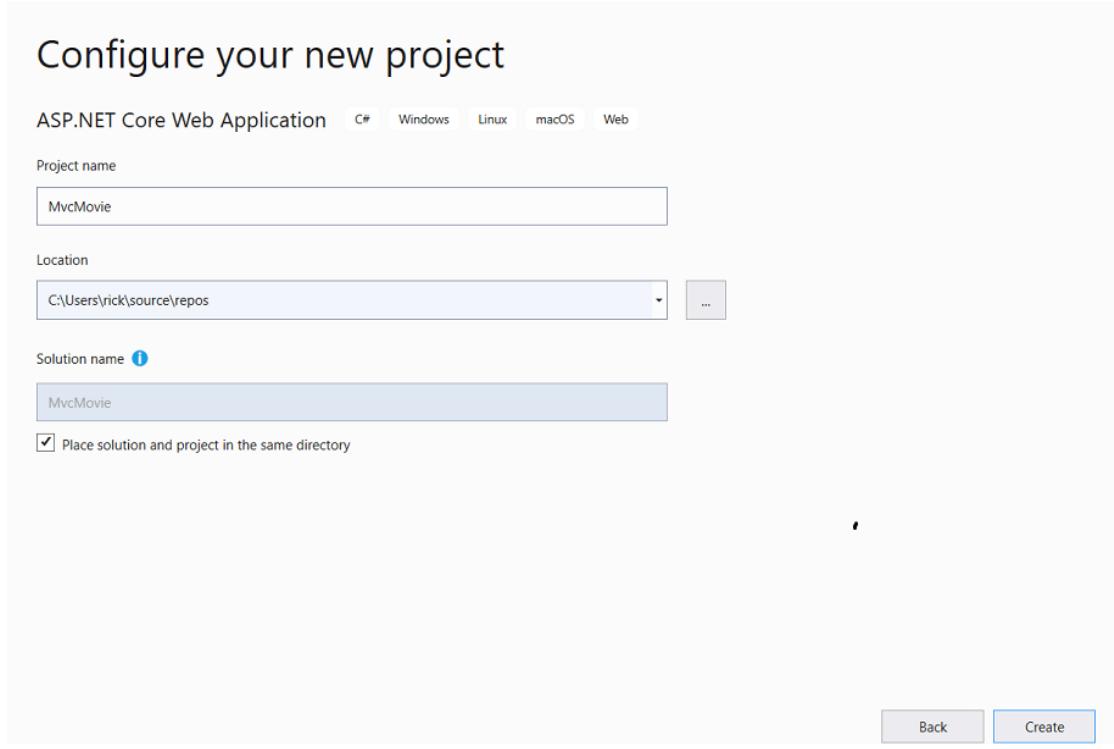
Create a web app

[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

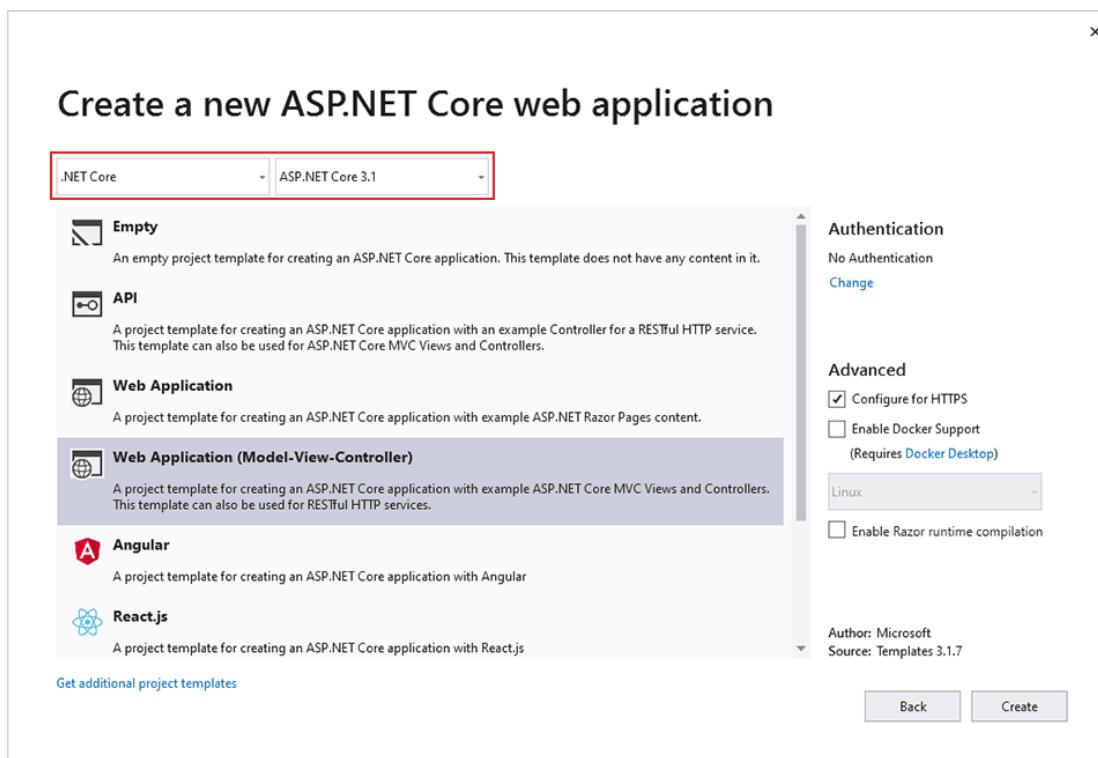
- From the Visual Studio, select **Create a new project**.
- Select **ASP.NET Core Web Application > Next**.



- Name the project **MvcMovie** and select **Create**. It's important to name the project **MvcMovie** so when you copy code, the namespace will match.



- Select **Web Application(Model-View-Controller)**. From the dropdown boxes, select **.NET Core** and **ASP.NET Core 3.1**, then select **Create**.



Visual Studio used the default project template for the created MVC project. The created project:

- Is a working app.

- Is a basic starter project.

Run the app

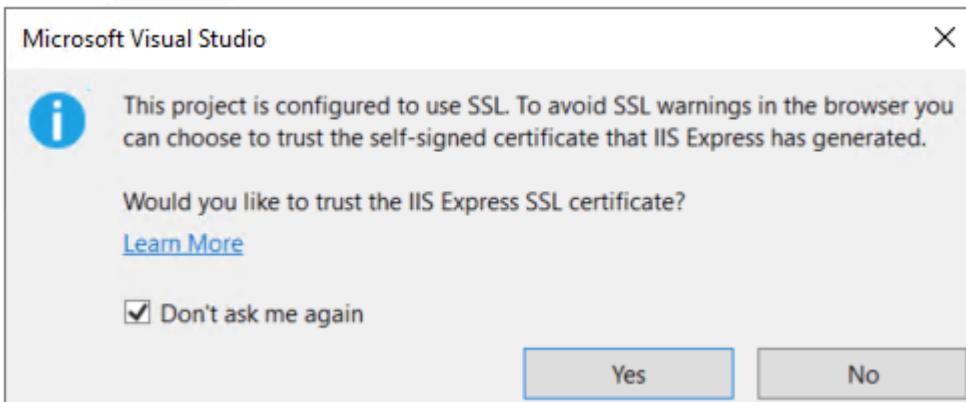
Visual Studio

Visual Studio Code

Visual Studio for Mac

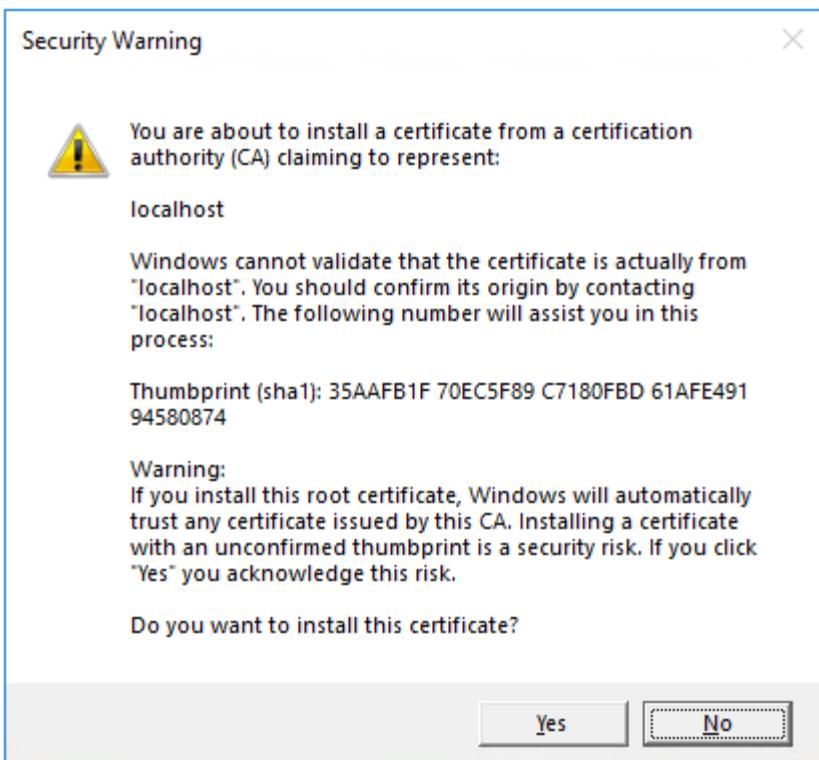
- Select Ctrl+F5 to run the app without debugging.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see [Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error](#).

Visual Studio:

- Starts [IIS Express](#).
- Runs the app.

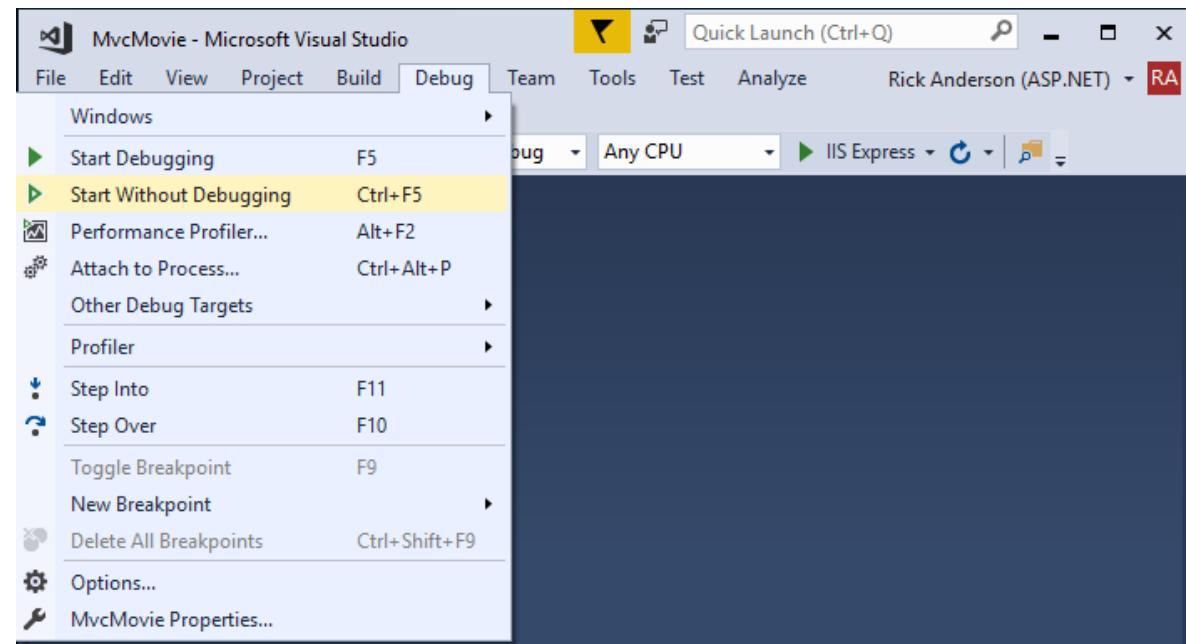
The address bar shows `localhost:port#` and not something like `example.com`.

The standard hostname for your local computer is `localhost`. When Visual Studio creates a web project, a random port is used for the web server.

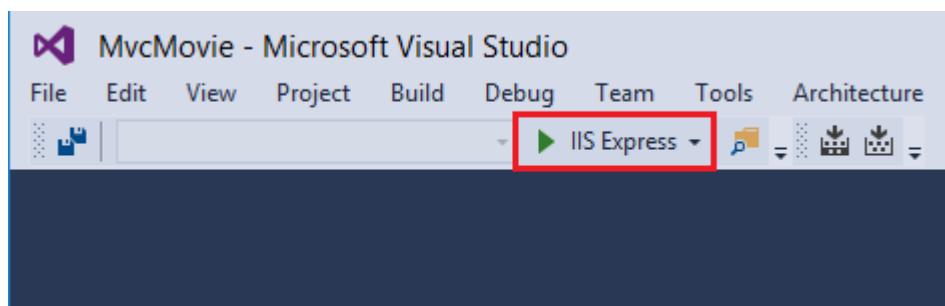
Launching the app without debugging by selecting **Ctrl+F5** allows you to:

- Make code changes.
- Save the file.
- Quickly refresh the browser and see the code changes.

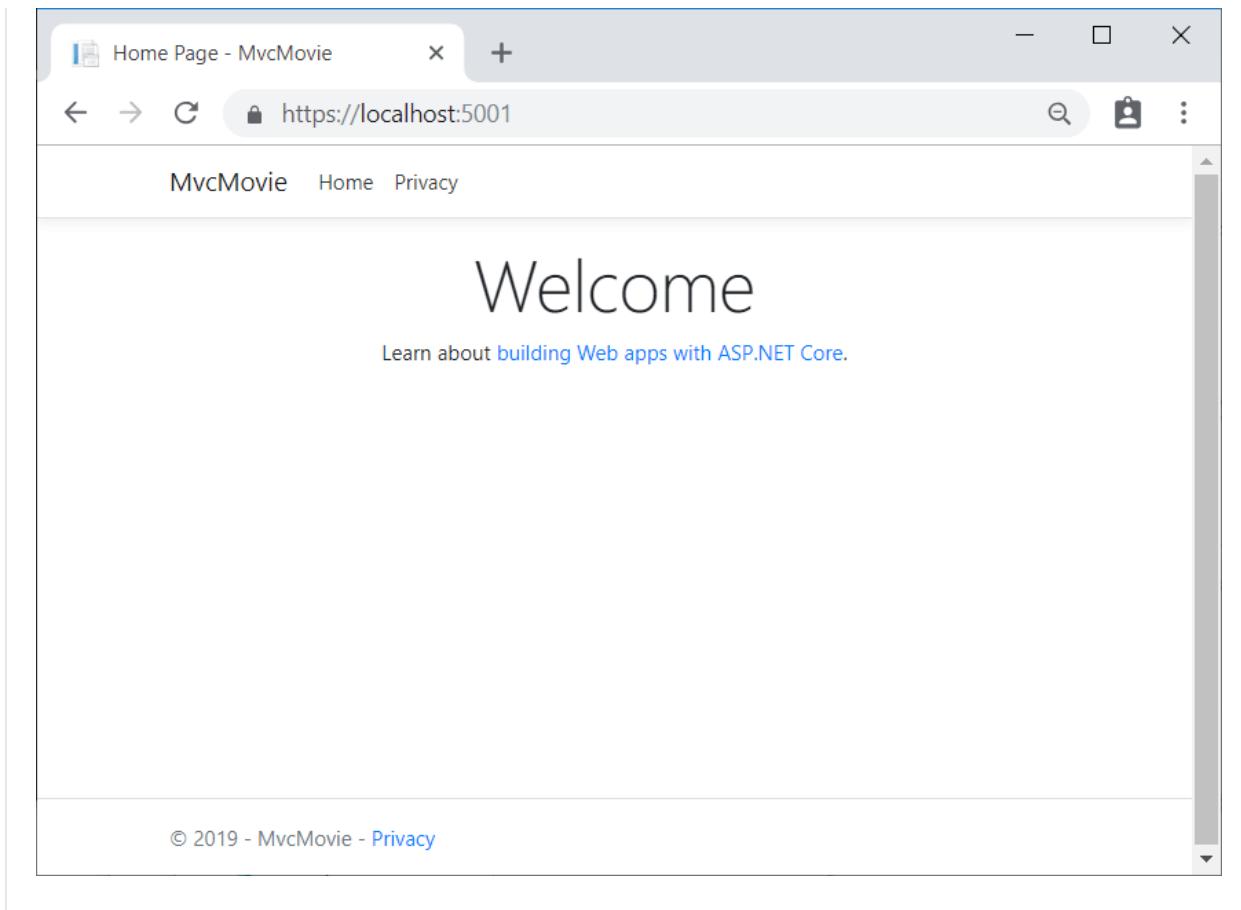
You can launch the app in debug or non-debug mode from the **Debug** menu item:



You can debug the app by selecting the IIS Express button



The following image shows the app:

[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

Visual Studio help

- [Learn to debug C# code using Visual Studio](#)
- [Introduction to the Visual Studio IDE](#)

In the next part of this tutorial, you learn about MVC and start writing some code.

[Next](#)

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

Article • 09/27/2021 • 11 minutes to read •  +13

[Is this page helpful?](#)

In this article

[Add a controller](#)

By [Rick Anderson](#)

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

MVC-based apps contain:

- **Models:** 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 tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **Views:** Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **Controllers:** 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://localhost:5001/Home/Privacy` : specifies the `Home` controller and the `Privacy` action.
- `https://localhost:5001/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.

Route data is explained 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 tutorial series while building a movie app. The MVC project contains folders for the `Controllers` and `Views`.

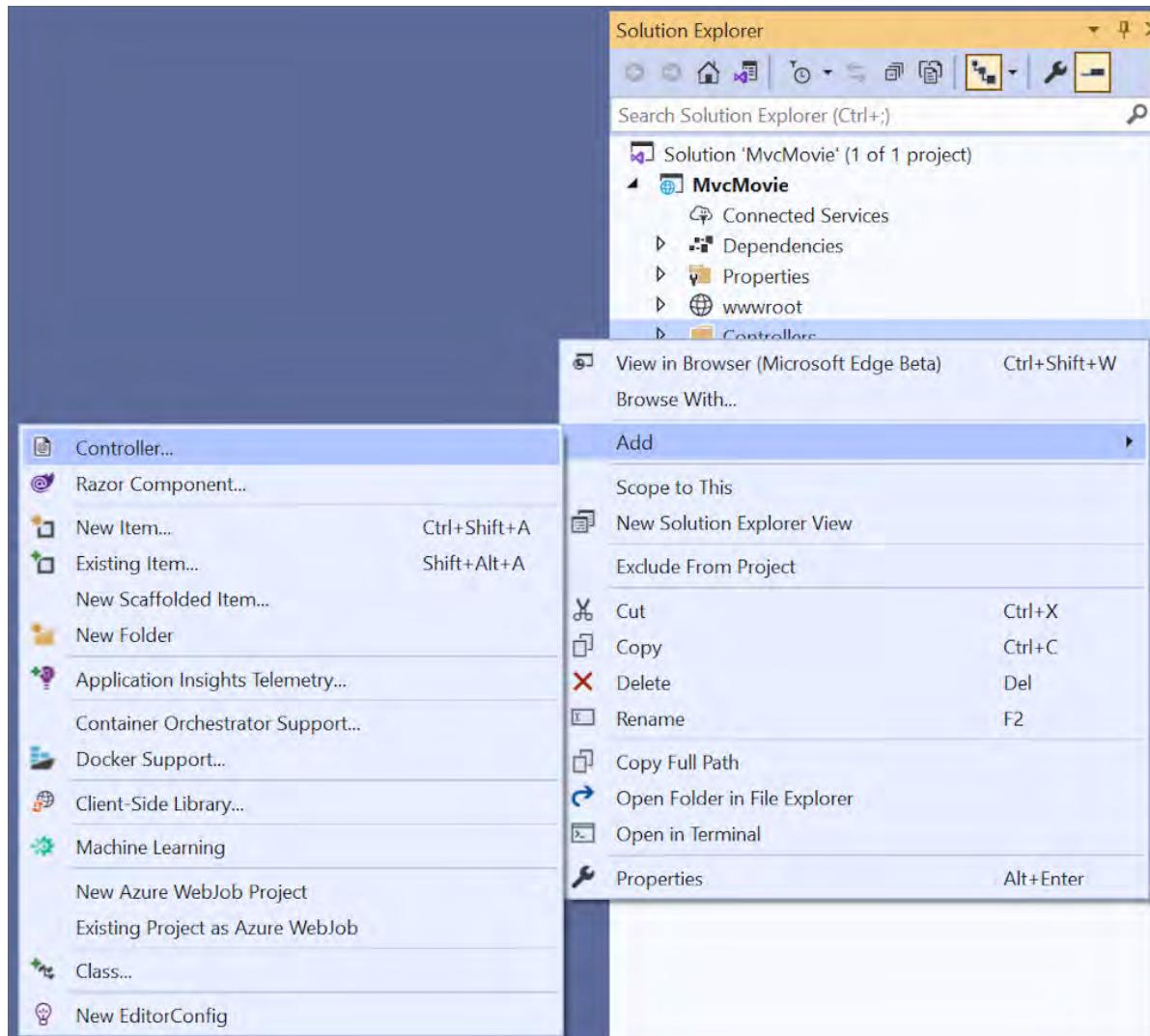
Add a controller

Visual Studio

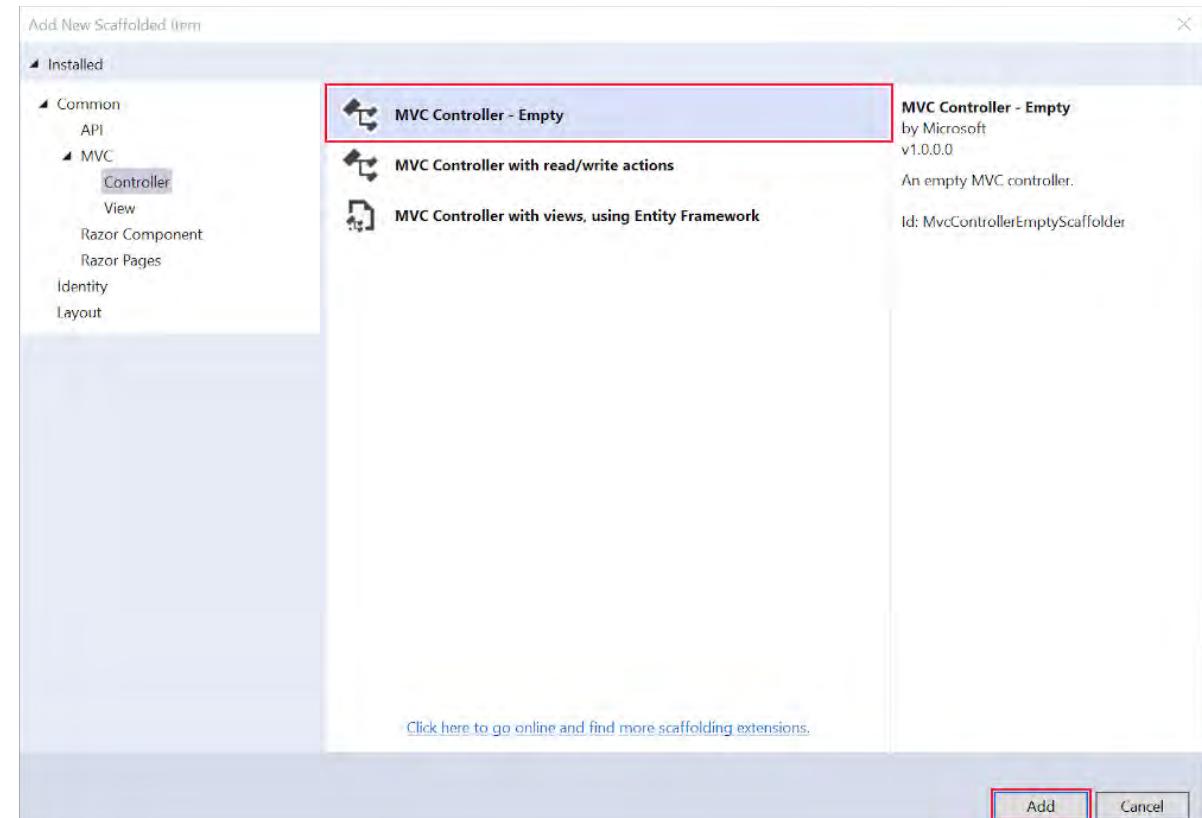
Visual Studio Code

Visual Studio for Mac

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



In the **Add Scaffold** dialog box, select **MVC Controller - Empty**.



In the Add New Item - MvcMovie dialog, enter `HelloWorldController.cs` and select Add.

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

```
C# Copy
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:

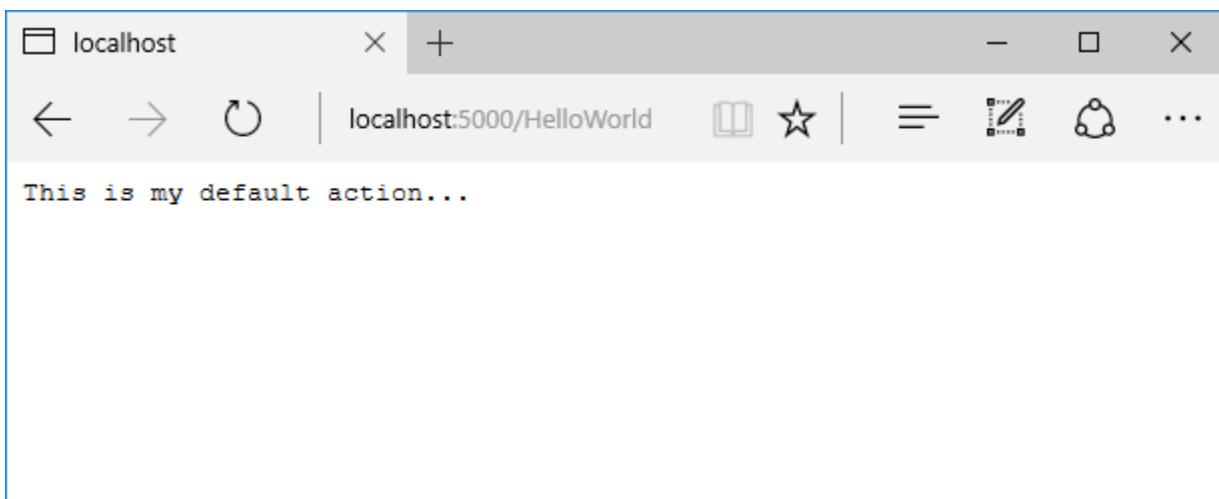
- Is a targetable URL in the web application, such as `https://localhost:5001 /HelloWorld`.
- Combines:
 - The protocol used: `HTTPS`.
 - The network location of the web server, including the TCP port: `localhost:5001`.
 - The target URI: `HelloWorld`.

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 `Configure` method in `Startup.cs` file.

C#

 Copy

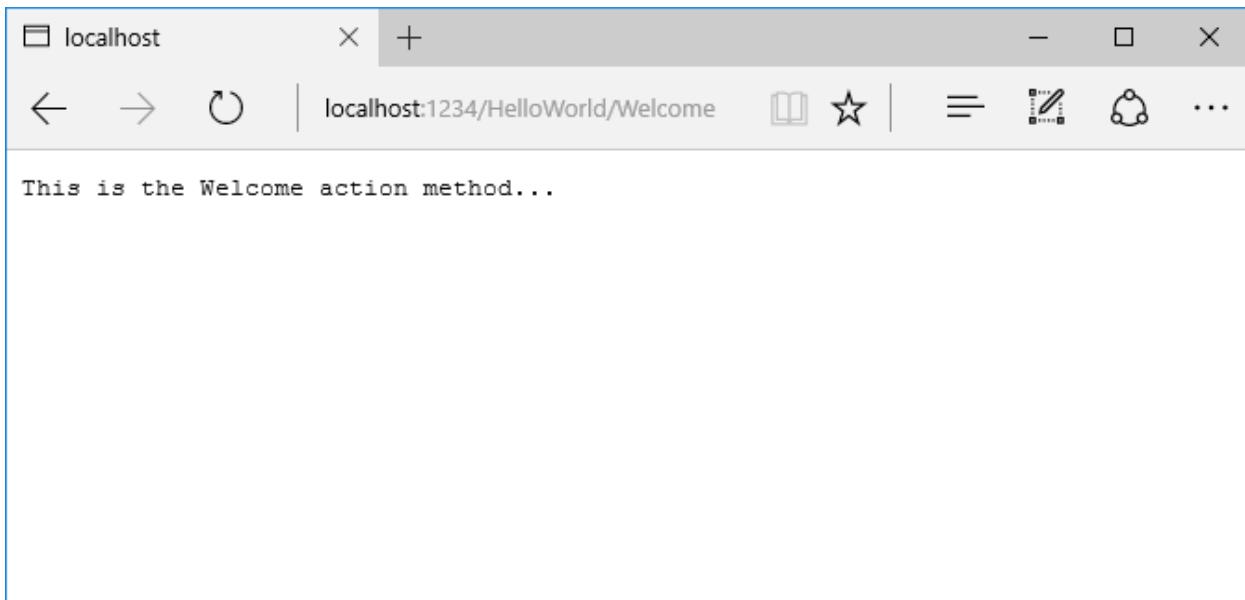
```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

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 `localhost:5001/HelloWorld` maps to the `HelloWorldController` class.
- The second part of the URL segment determines the action method on the class. So `localhost:5001/HelloWorld/Index` causes the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:5001/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://localhost:{PORT}/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.

C#

Copy

```
// 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://localhost:{PORT}/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:

C#	 Copy
<pre>public string Welcome(string name, int ID = 1) { return HtmlEncoder.Default.Encode(\$"Hello {name}, ID: {ID}"); }</pre>	

Run the app and enter the following URL: `https://localhost:{PORT}/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](#) .

C#

 Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.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.

[Previous: Get Started](#)[Next: Add a View](#)

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

Article • 09/27/2021 • 16 minutes to read •  +14

[Is this page helpful?](#)

In this article

[Add a view](#)

[Change views and layout pages](#)

[Change the title, footer, and menu link in the layout file](#)

[Passing Data from the Controller to the View](#)

By [Rick Anderson](#)

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:

C#	 Copy
<pre>public IActionResult Index() { return View(); }</pre>	

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

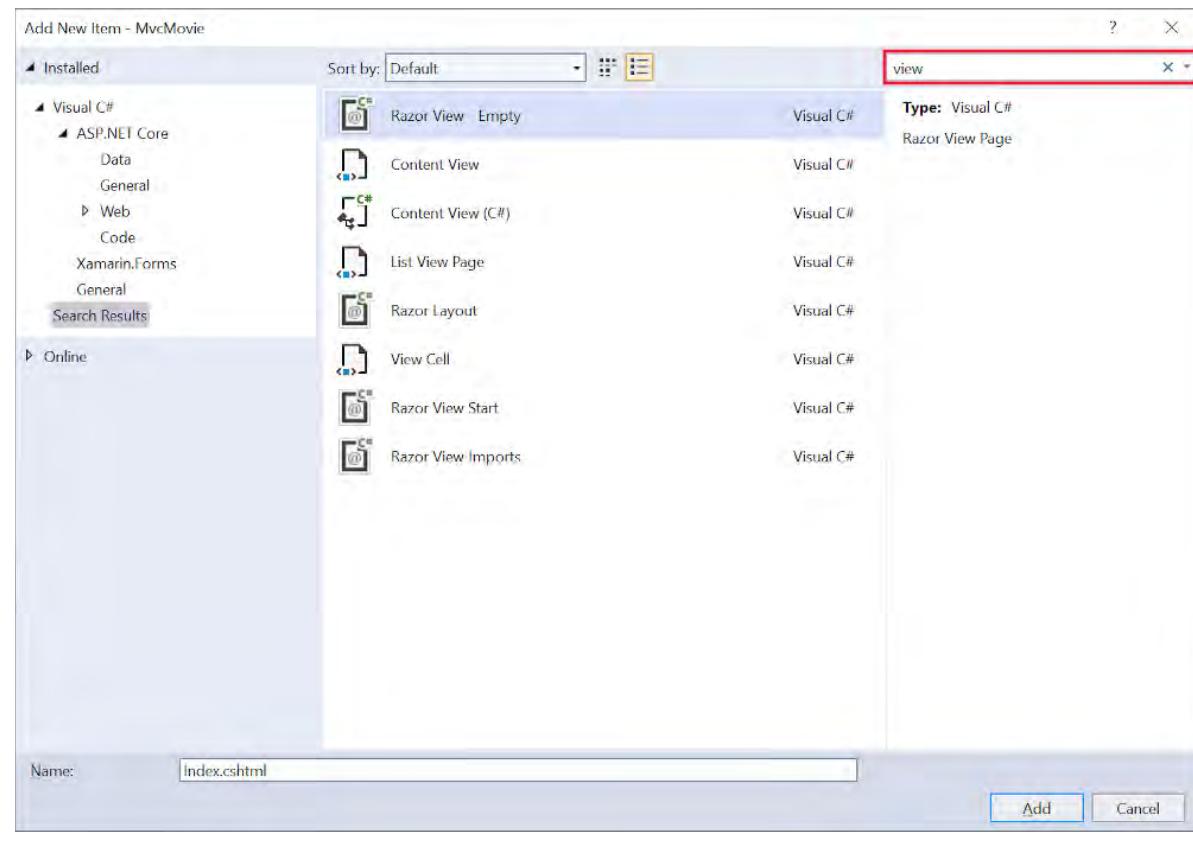
[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

Right-click on the `Views` folder, and then **Add > New Folder** and name the folder `HelloWorld`.

Right-click on the `Views/HelloWorld` folder, and then **Add > New Item**.

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter *view*
- Select **Razor View - Empty**
- Keep the **Name** box value, `Index.cshtml`.
- Select **Add**

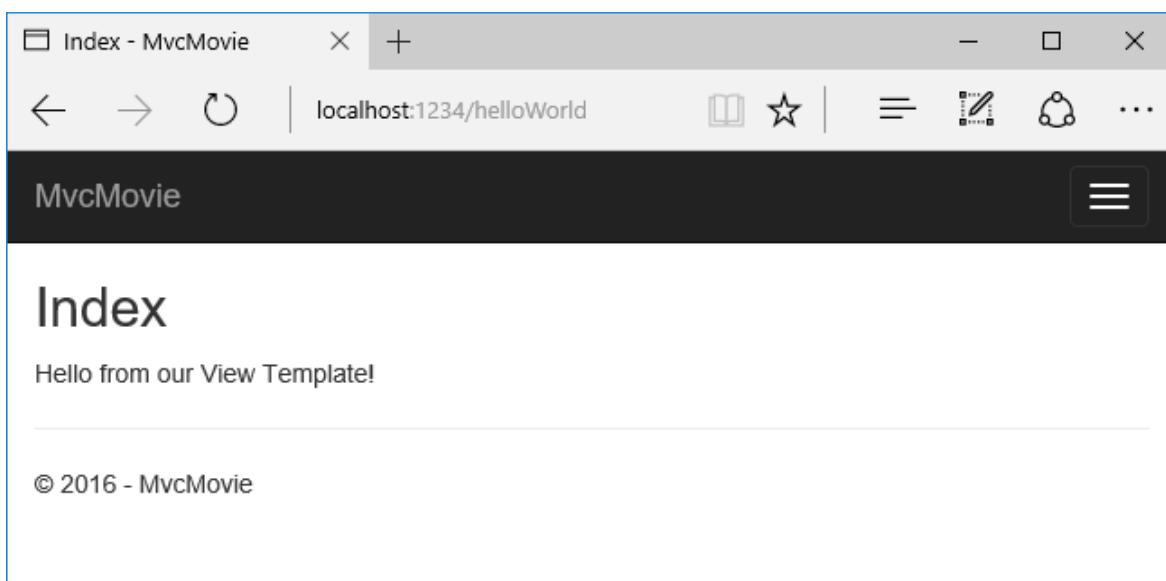


Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

CSHTML	 Copy
<pre>@{ ViewData["Title"] = "Index"; } <h2>Index</h2> <p>Hello from our View Template!</p></pre>	

Navigate to `https://localhost:{PORT}/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 allows:

- 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. The changes are highlighted:

CSHTML	 Copy
<pre><!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.min.css" /> <link rel="stylesheet" href="~/css/site.css" /> </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"> Movie App <button class="navbar-toggler" type="button" data-tog-</pre>	

```
gle="collapse" data-target=".navbar-collapse" aria-con-
trols="navbarSupportedContent"
            aria-expanded="false" aria-label="Toggle naviga-
tion">
    <span class="navbar-toggler-icon"></span>
</button>
<div class="navbar-collapse collapse d-sm-inline-flex flex-
sm-row-reverse">
    <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>
</header>
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>

<footer class="border-top footer text-muted">
    <div class="container">
        &copy; 2020 - Movie App - <a asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
    </div>
</footer>
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@RenderSection("Scripts", required: false)
</body>
</html>
```

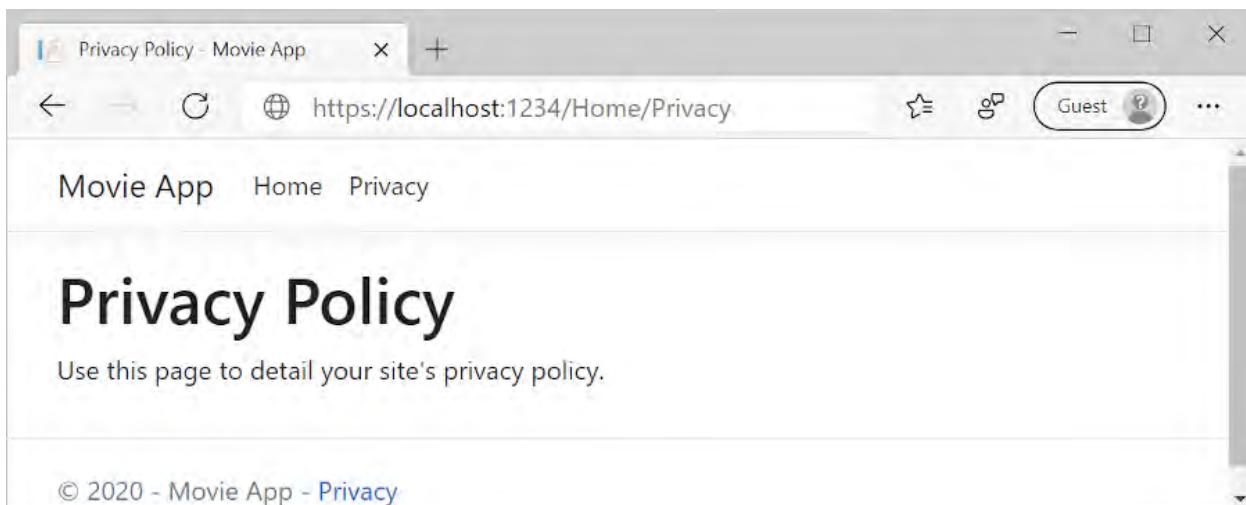
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `MvcMovies` to `Movie App`.

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 - Mvc Movie**:



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:

CSHTML	Copy
<pre>@{ Layout = "_Layout"; }</pre>	

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:

CSHTML

 Copy

```
@{  
    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:

CSHTML

 Copy

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

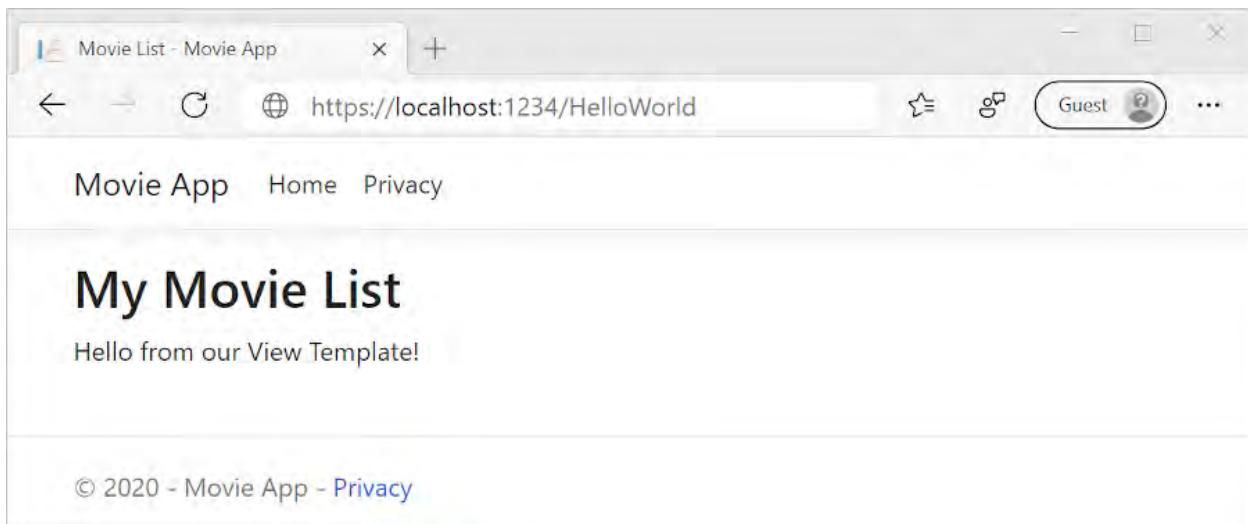
Save the change and navigate to `https://localhost:{PORT}/HelloWorld`.

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 a `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`:

C#	 Copy
<pre>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(); } } }</pre>	

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`.

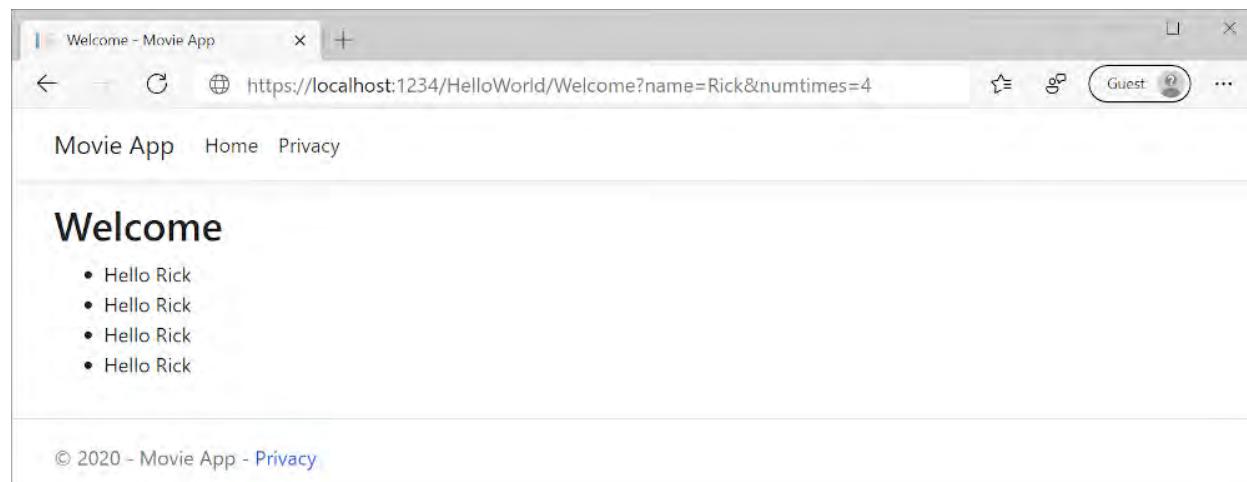
Replace the contents of `Views/HelloWorld/Welcome.cshtml` with the following:

CSHTML	 Copy
<pre>@{ ViewData["Title"] = "Welcome"; } <h2>Welcome</h2> @for (int i = 0; i < (int)ViewData["NumTimes"]; i++) { @ViewData["Message"] } </pre>	

Save your changes and browse to the following URL:

`https://localhost:{PORT}/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.

In the next tutorial, a database of movies is created.

[Previous: Add a Controller](#)

[Next: Add a Model](#)

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

Article • 11/19/2021 • 42 minutes to read •  +18

[Is this page helpful?](#)

In this article

- [Add a data model class](#)
- [Add NuGet packages](#)
- [Create a database context class](#)
- [Register the database context](#)
- [Examine the database connection string](#)
- [Scaffold movie pages](#)
- [Initial migration](#)
- [Test the app](#)
- [Dependency injection in the controller](#)
- [Strongly typed models and the @model keyword](#)
- [Additional resources](#)

By [Rick Anderson](#) and [Jon P Smith](#).

In this tutorial, classes are added for managing movies in a database. These classes are the "Model" part of the MVC app.

These model classes are used 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 created are known as **POCO** classes, from **Plain Old CLR Objects**. 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 tutorial, model classes are created 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](#)

```
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.

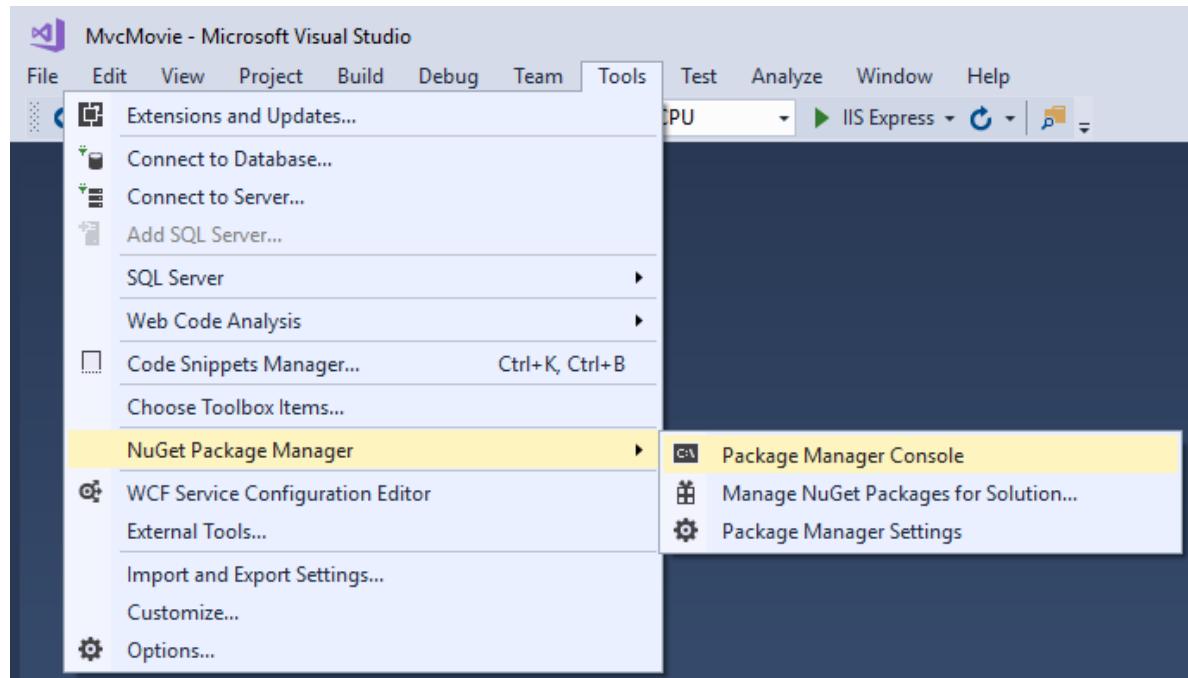
[DataAnnotations](#) are covered in a later tutorial.

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:

```
PowerShell
```

Copy

```
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

```
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 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
<pre>using MvcMovie.Data; using Microsoft.EntityFrameworkCore;</pre>	

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.

Examine the database connection string

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

Visual Studio

Visual Studio Code / Visual Studio for Mac

JSON

 Copy

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

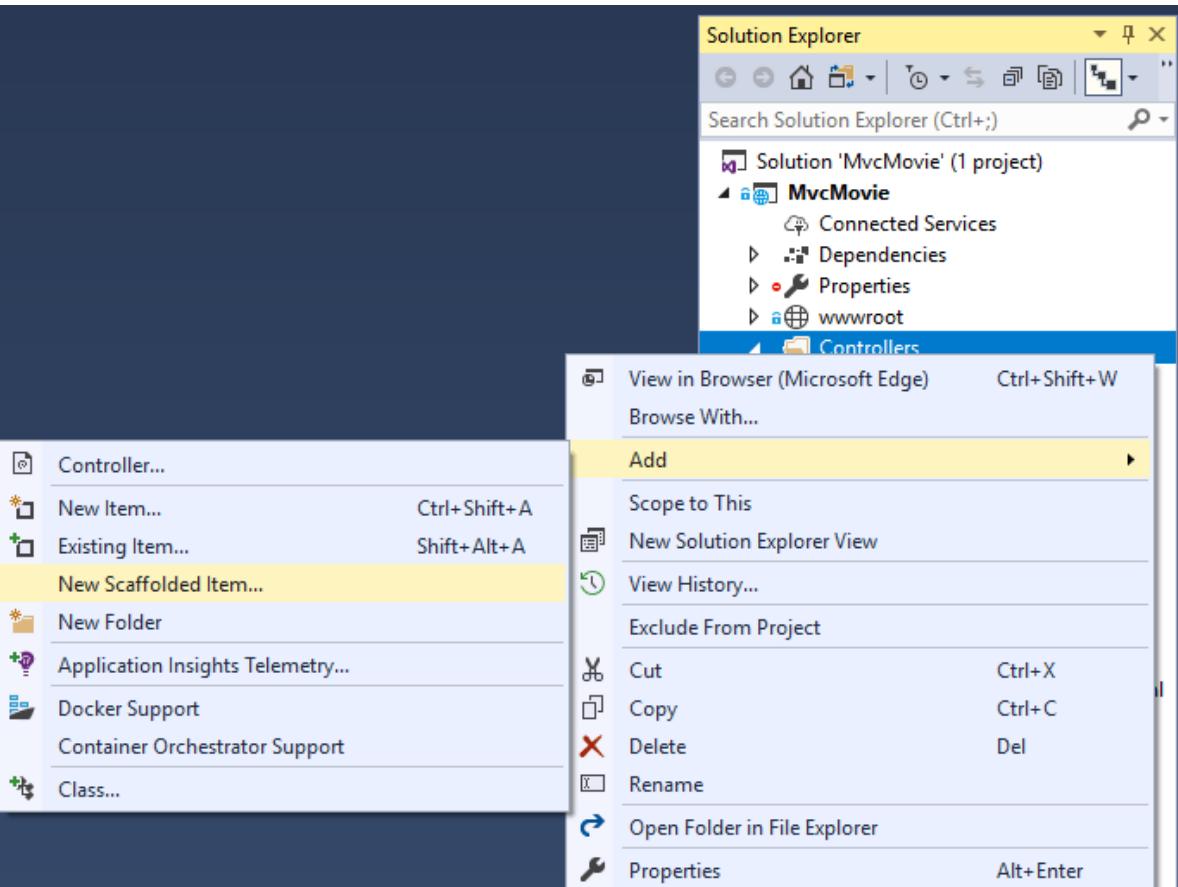
Build the project as a check for compiler errors.

Scaffold movie pages

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

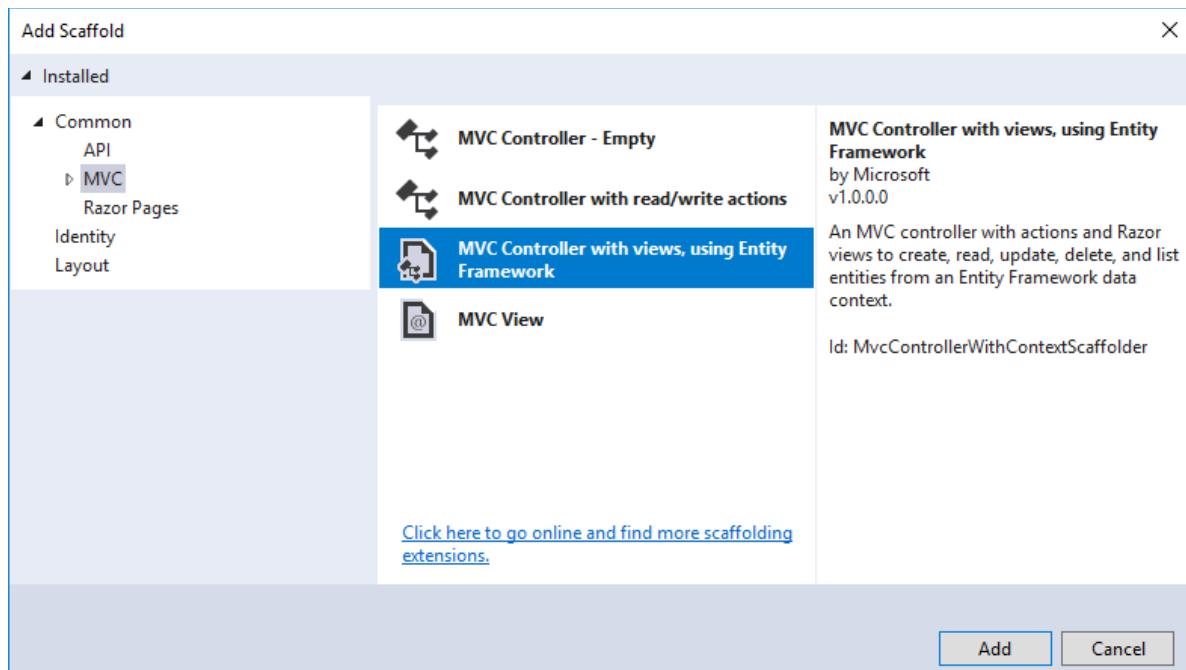
Visual Studio Visual Studio Code Visual Studio for Mac

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



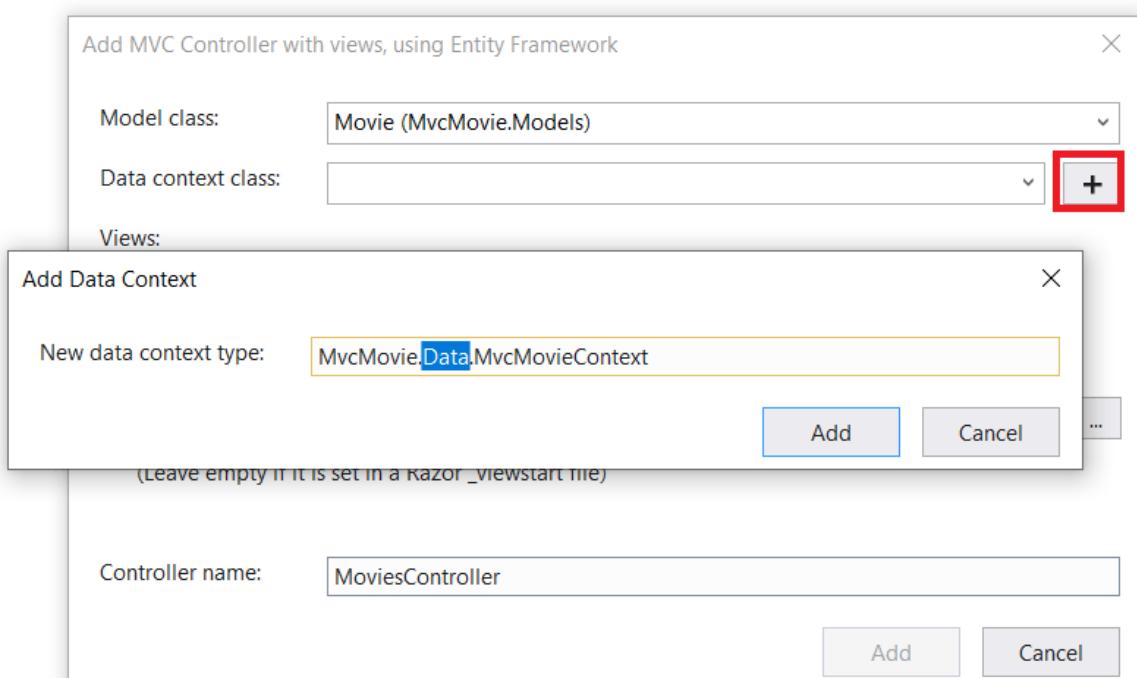
The screenshot shows the Visual Studio interface with the Solution Explorer window open. The 'Controllers' folder is selected in the tree view. A context menu is open over the folder, with the 'Add' option highlighted. Other options visible in the menu include 'View in Browser (Microsoft Edge)', 'Scope to This', 'New Solution Explorer View', 'View History...', 'Exclude From Project', 'Cut', 'Copy', 'Delete', 'Rename', 'Open Folder in File Explorer', and 'Properties'. On the left, a sidebar lists various scaffolded item options like 'Controller...', 'New Item...', 'Existing Item...', etc.

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



Complete the Add Controller dialog:

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



- **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:

PowerShell

 Copy

```
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
<pre>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"), 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(</pre>	

```
        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:

A screenshot of a terminal window. At the top, there are two tabs: "Visual Studio" (which is selected) and "Visual Studio Code / Visual Studio for Mac". Below the tabs, the console output shows the following text:
Console
Copy
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

```
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

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

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

CSHTML

 Copy

```
@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:

CSHTML

 Copy

```
@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:

CSHTML

 Copy

```
@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:

CSHTML

 Copy

```
@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, this means that you get compile time checking of the code.

Additional resources

- [Tag Helpers](#)
- [Globalization and localization](#)

[Previous Adding a View](#)[Next Working with SQL](#)

Part 5, work with a database in an ASP.NET Core MVC app

Article • 09/27/2021 • 9 minutes to read •  +6

[Is this page helpful?](#)

In this article

[SQL Server Express LocalDB](#)

[Seed the database](#)

By [Rick Anderson](#) and [Jon P Smith](#).

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the [Dependency Injection](#) container in the `ConfigureServices` method in the `Startup.cs` file:

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 ASP.NET Core [Configuration](#) system reads the `ConnectionString` key. For local development, it gets the connection string from the `appsettings.json` file:

JSON

 Copy

```
"ConnectionStrings": {  
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;  
  Database=MvcMovieContext-2;Trusted_Connection=True;  
  MultipleActiveResultSets=true"  
}
```

When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a production SQL Server. For more information, see [Configuration](#).

Visual Studio

Visual Studio Code / Visual Studio for Mac

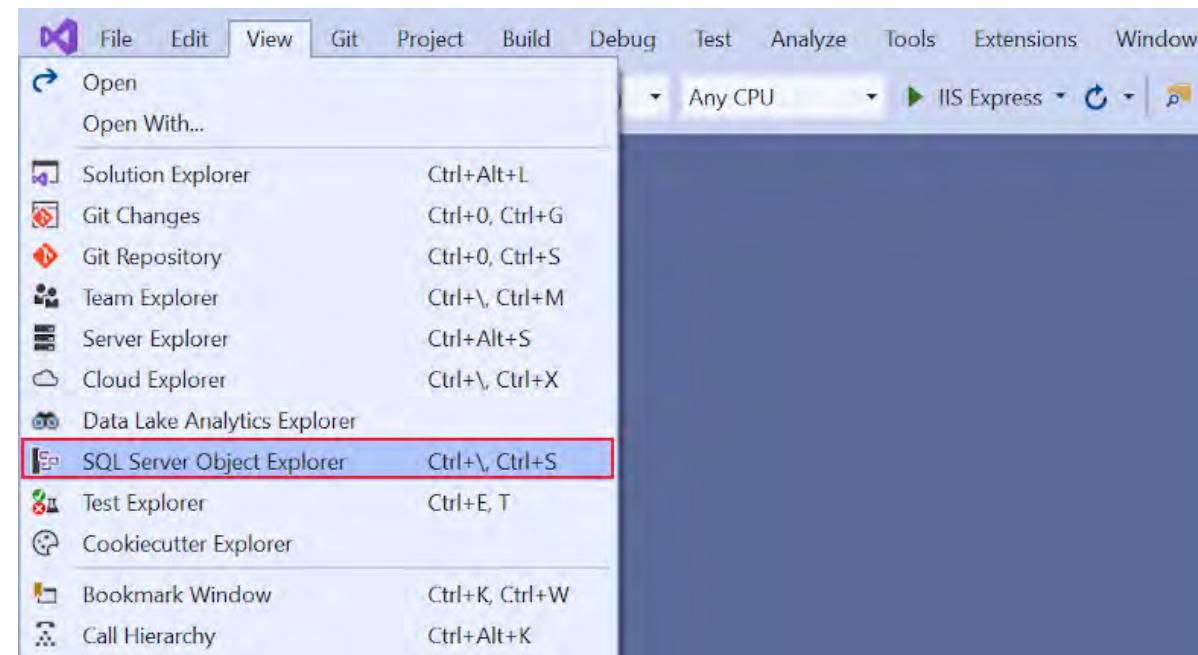
SQL Server Express LocalDB

LocalDB:

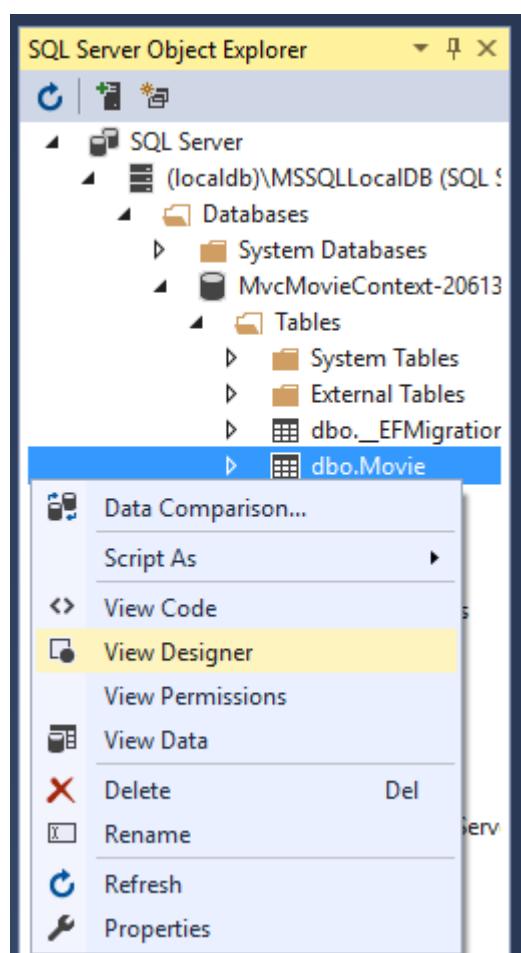
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Examine the database

From the **View** menu, open **SQL Server Object Explorer** (SSOX).



Right-click on the Movie table > View Designer



The screenshot shows the SSMS 'Design' tab for the 'dbo.Movie' table. The table structure is as follows:

	Name	Data Type	Allow Nulls
PK	ID	int	<input type="checkbox"/>
	Genre	nvarchar(MAX)	<input checked="" type="checkbox"/>
	Price	decimal(18,2)	<input type="checkbox"/>
	ReleaseDate	datetime2(7)	<input type="checkbox"/>
	Title	nvarchar(MAX)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

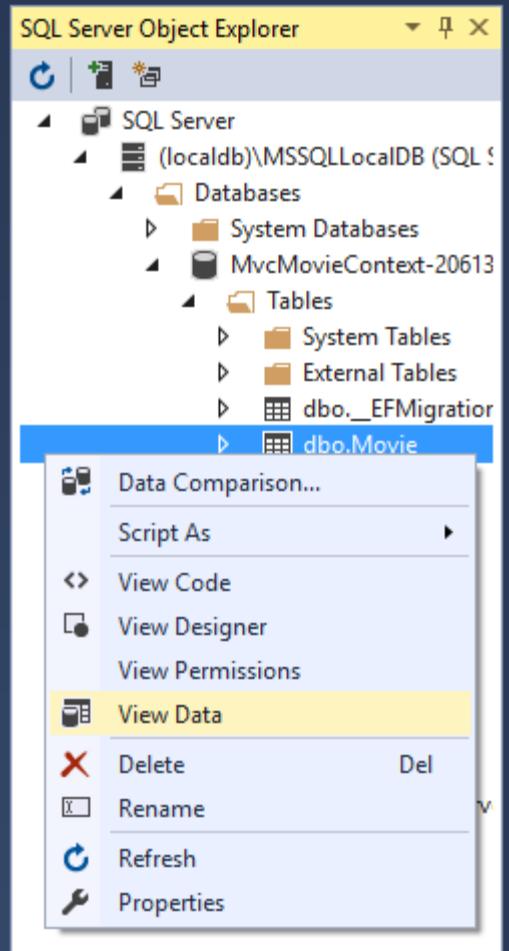
On the right, there are sections for Keys (1), Check Constraints (0), Indexes (0), Foreign Keys (0), and Triggers (0). Below the table structure, the T-SQL script pane contains the following code:

```
1 CREATE TABLE [dbo].[Movie] (
2     [ID] INT IDENTITY (1, 1) NOT NULL,
3     [Genre] NVARCHAR (MAX) NULL,
4     [Price] DECIMAL (18, 2) NOT NULL,
5     [ReleaseDate] DATETIME2 (7) NOT NULL,
6     [Title] NVARCHAR (MAX) NULL,
7     CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
8 );
9
```

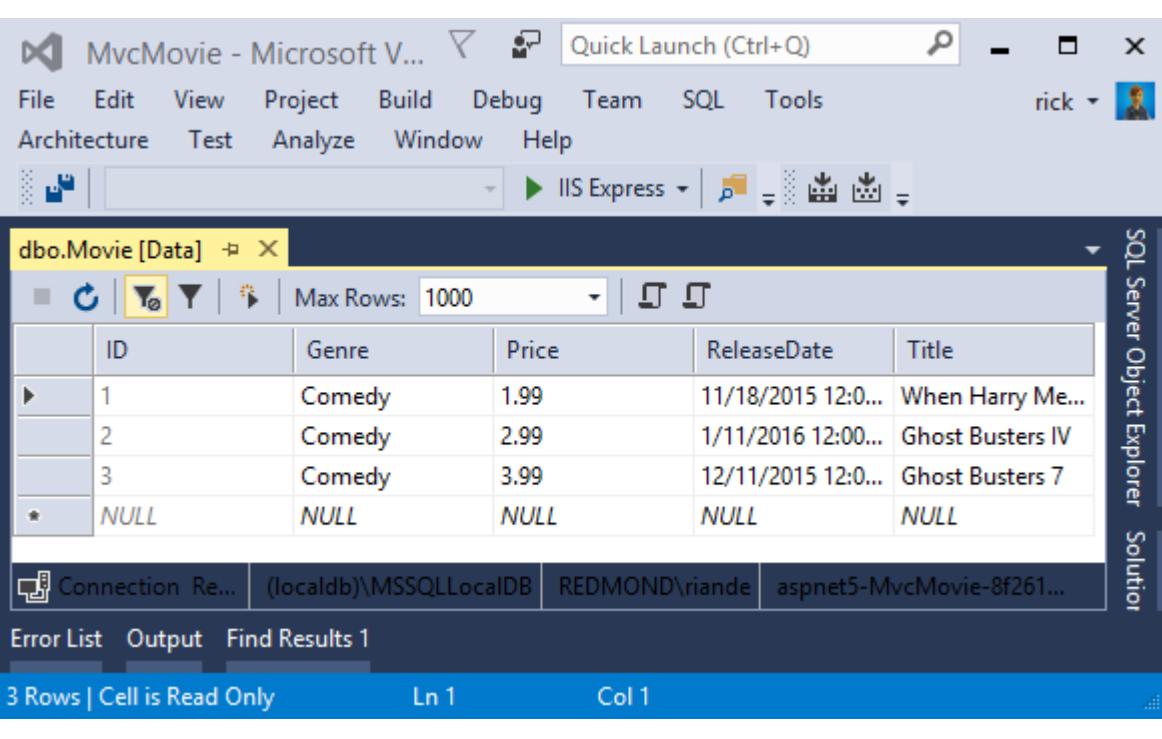
The status bar at the bottom shows 'Connection Ready' and the connection details: (localdb)\MSSQLLocalDB | REDMOND\riande | aspnet5-MvcMovie-8f261...

Note the key icon next to `ID`. By default, EF makes a property named `ID` the primary key.

Right-click on the `Movie` table > `View Data`



The screenshot shows the SQL Server Object Explorer window. The tree view on the left shows the database structure: SQL Server, (localdb)\MSSQLLocalDB (SQL Server), Databases, System Databases, MvcMovieContext-20613, Tables, System Tables, External Tables, dbo._EFMigration, and dbo.Movie. The 'dbo.Movie' table is selected and highlighted with a blue selection bar. A context menu is open over the table, listing options: Data Comparison..., Script As, View Code, View Designer, View Permissions, View Data (which is highlighted with a yellow selection bar), Delete, Rename, Refresh, and Properties.



The screenshot shows the Microsoft Visual Studio interface with the title bar 'MvcMovie - Microsoft V...'. The main area displays a data grid titled 'dbo.Movie [Data]'. The grid has columns: ID, Genre, Price, ReleaseDate, and Title. The data rows are:

ID	Genre	Price	ReleaseDate	Title
1	Comedy	1.99	11/18/2015 12:00:00 AM	When Harry Met...
2	Comedy	2.99	1/11/2016 12:00:00 AM	Ghost Busters IV
3	Comedy	3.99	12/11/2015 12:00:00 AM	Ghost Busters 7
*	NULL	NULL	NULL	NULL

Below the grid, the status bar shows '3 Rows | Cell is Read Only' and 'Ln 1 Col 1'. The bottom navigation bar includes tabs for Error List, Output, Find Results 1, and a connection status bar showing '(localdb)\MSSQLLocalDB', 'REDMOND\riande', and 'aspnet5-MvcMovie-8f261...'. The right sidebar shows 'SQL Server Object Explorer' and 'Solution'.

Seed the database

Create a new class named `SeedData` in the `Models` folder. Replace the generated code with the following:

C#	 Copy
----	--

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                    }
                );
            }
        }
    }
}
```

```
        Price = 9.99M
    },

    new Movie
    {
        Title = "Rio Bravo",
        ReleaseDate = DateTime.Parse("1959-4-15"),
        Genre = "Western",
        Price = 3.99M
    }
);
context.SaveChanges();
}
}
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

C#

 Copy

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of *Program.cs* with the following code:

C#

 Copy

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using MvcMovie.Data;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
```

```
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Test the app.

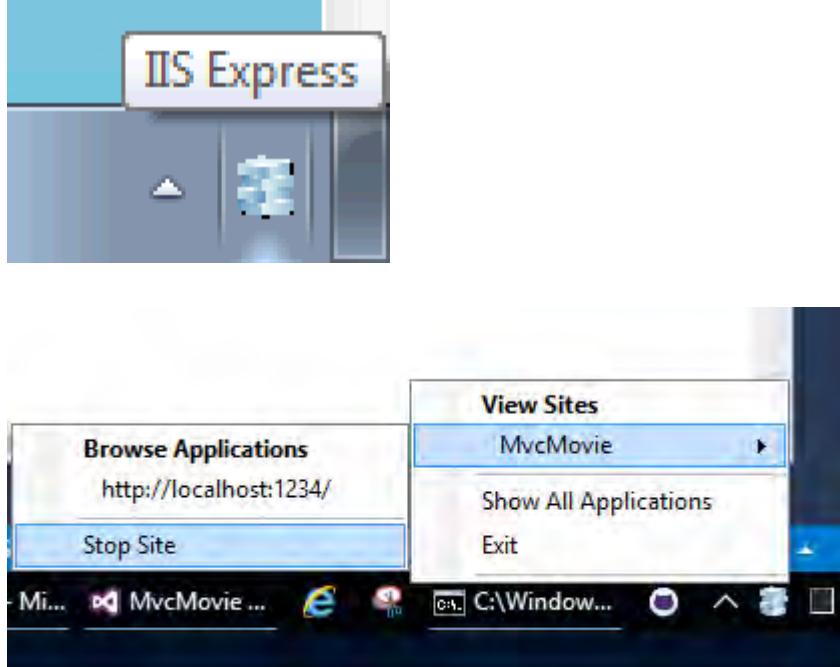
Visual Studio

Visual Studio Code / Visual Studio for Mac

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

Force the app to initialize, calling the methods in the `Startup` class, so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

- Right-click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



- If you were running VS in non-debug mode, press F5 to run in debug mode
- If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

© 2017 - MvcMovie

[Previous: Adding a model](#)[Next: Adding controller methods and views](#)

Part 6, controller methods and views in ASP.NET Core

Article • 09/27/2021 • 16 minutes to read •  +13

[Is this page helpful?](#)

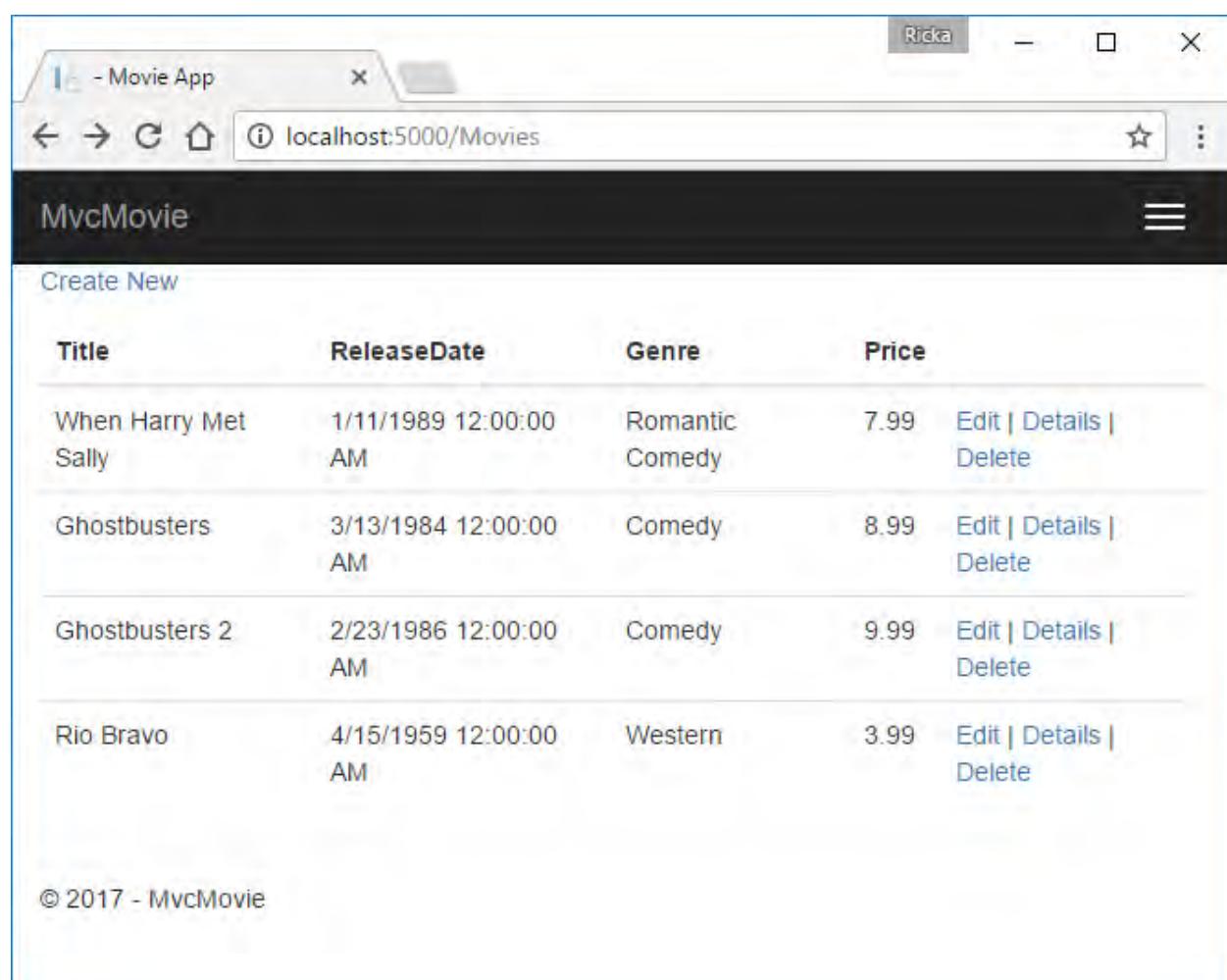
In this article

[Processing the POST Request](#)

[Additional resources](#)

By [Rick Anderson](#)

We have a good start to the movie app, but the presentation isn't ideal, for example, `ReleaseDate` should be two words.



Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

© 2017 - MvcMovie

Open the `Models/Movie.cs` file and add the highlighted lines shown below:

C#

 Copy

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

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

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

We cover [DataAnnotations](#) in the next tutorial. The [Display](#) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](#) attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see [Data Types](#).

Browse to the `Movies` controller and hold the mouse pointer over an [Edit](#) link to see the target URL.

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the *Views/Movies/Index.cshtml* file.

CSHTML	Copy
<pre><a asp-action="Edit" asp-route-id="@item.ID">Edit <a asp-action="Details" asp-route-id="@item.ID">Details <a asp-action="Delete" asp-route-id="@item.ID">Delete </td> </tr></pre>	

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

HTML	Copy
<pre><td></pre>	

```
<a href="/Movies/Edit/4"> Edit </a> |
<a href="/Movies/Details/4"> Details </a> |
<a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for **routing** set in the *Startup.cs* file:

C#

 Copy

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

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see [Additional resources](#).

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.

C#

 Copy

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

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

C#	 Copy
<pre>// POST: Movies/Edit/5 // To protect from overposting attacks, please enable the specific properties you want to bind to, for // more details see http://go.microsoft.com/fwlink/?LinkId=317598. [HttpPost] [ValidateAntiForgeryToken] public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie) { if (id != movie.ID) { return NotFound(); } if (ModelState.IsValid) { try { _context.Update(movie); await _context.SaveChangesAsync(); } catch (DbUpdateConcurrencyException) { if (!MovieExists(movie.ID)) { return NotFound(); } else { throw; } } return RedirectToAction("Index"); } return View(movie); }</pre>	

The `[Bind]` attribute is one way to protect against [over-posting](#). You should only include properties in the `[Bind]` attribute that you want to change. For more information, see [Protect your controller from over-posting](#). [ViewModels](#) provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

C#

 Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to [prevent forgery of a request](#) and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token with the [Form Tag Helper](#).

CSHTML

 Copy

```
<form asp-action="Edit">
```

The [Form Tag Helper](#) generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the Movies controller. For more information, see [Prevent Cross-Site Request Forgery \(XSRF/CSRF\) attacks in ASP.NET Core](#).

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the `Edit` view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

C#

 Copy

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

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the `Edit` view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the `Edit` view that was generated by the Visual Studio scaffolding system:

CSHTML

 Copy

```
@model MvcMovie.Models.Movie

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

```
<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
                </div>
                <input type="hidden" asp-for="Id" />
                <div class="form-group">
                    <label asp-for="Title" class="control-label"></label>
                    <input asp-for="Title" class="form-control" />
                    <span asp-validation-for="Title" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="ReleaseDate" class="control-label"></label>
                    <input asp-for="ReleaseDate" class="form-control" />
                    <span asp-validation-for="ReleaseDate" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Genre" class="control-label"></label>
                    <input asp-for="Genre" class="form-control" />
                    <span asp-validation-for="Genre" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Price" class="control-label"></label>
                    <input asp-for="Price" class="form-control" />
                    <span asp-validation-for="Price" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <input type="submit" value="Save" class="btn btn-primary" />
                </div>
            </form>
        </div>
    </div>
<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top

of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The [Label Tag Helper](#) displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The [Input Tag Helper](#) renders an HTML `<input>` element. The [Validation Tag Helper](#) displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an [Edit](#) link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

HTML	 Copy
<pre><form action="/Movies/Edit/7" method="post"> <div class="form-horizontal"> <h4>Movie</h4> <hr /> <div class="text-danger" /> <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID" value="7" /> <div class="form-group"> <label class="control-label col-md-2" for="Genre" /> <div class="col-md-10"> <input class="form-control" type="text" id="Genre" name="Genre" value="Western" /> </div> </div> <div class="form-group"> <label class="control-label col-md-2" for="Price" /> <div class="col-md-10"> <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" /> </div> </div> <!-- Markup removed for brevity --> <div class="form-group"> <div class="col-md-offset-2 col-md-10"> <input type="submit" value="Save" class="btn btn-default" /> </div> </div> </div> </form></pre>	

```
</div>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCI1SduCRx9jDQC1rV9pOTTmq
UyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the Save button is clicked. The last line before the closing `</form>` element shows the hidden [XSRF](#) token generated by the [Form Tag Helper](#).

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

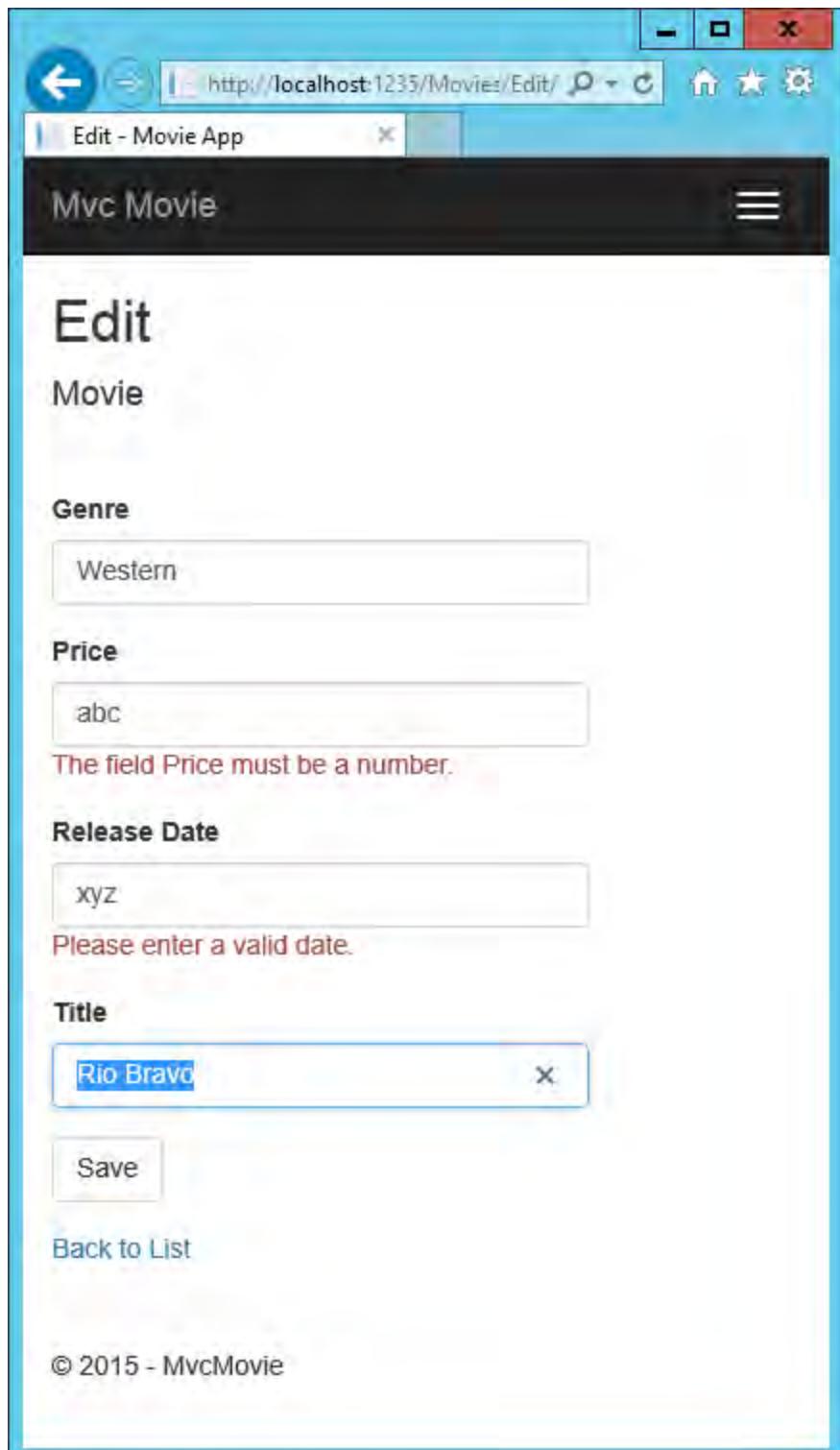
C#	 Copy
<pre>[HttpPost] [ValidateAntiForgeryToken] public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie) { if (id != movie.ID) { return NotFound(); } if (ModelState.IsValid) { try { _context.Update(movie); await _context.SaveChangesAsync(); } catch (DbUpdateConcurrencyException) { if (!MovieExists(movie.ID)) { return NotFound(); } else { throw; } } } }</pre>	

```
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden `XSRF` token generated by the anti-forgery token generator in the [Form Tag Helper](#)

The [model binding](#) system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` property verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine [Model Validation](#) in more detail. The [Validation Tag Helper](#) in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates `HTTP` best practices and the

architectural [REST](#) pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Author Tag Helpers](#)
- [Prevent Cross-Site Request Forgery \(XSRF/CSRF\) attacks in ASP.NET Core](#)
- Protect your controller from [over-posting](#)
- [ViewModels](#)
- [Form Tag Helper](#)
- [Input Tag Helper](#)
- [Label Tag Helper](#)
- [Select Tag Helper](#)
- [Validation Tag Helper](#)

[Previous](#)[Next](#)

Part 7, add search to an ASP.NET Core MVC app

Article • 11/19/2021 • 14 minutes to read •  +11

[Is this page helpful?](#)

In this article

[Add Search by genre](#)

[Add search by genre to the Index view](#)

By [Rick Anderson](#)

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside `Controllers/MoviesController.cs` with the following code:

C#	 Copy
<pre>public async Task<IActionResult> Index(string searchString) { var movies = from m in _context.Movie select m; if (!String.IsNullOrEmpty(searchString)) { movies = movies.Where(s => s.Title.Contains(searchString)); } return View(await movies.ToListAsync()); }</pre>	

The first line of the `Index` action method creates a [LINQ](#) query to select the movies:

C#	 Copy
<pre>var movies = from m in _context.Movie select m;</pre>	

The query is *only* defined at this point, it has **not** been run against the database.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

C#

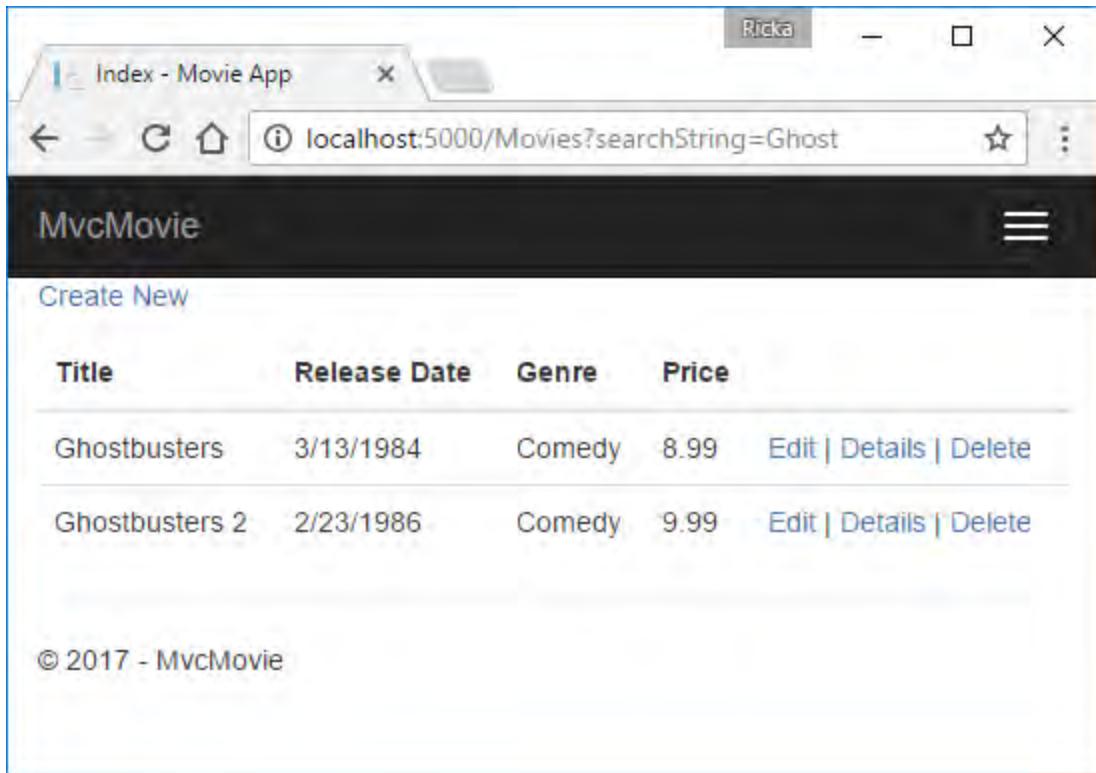
 Copy

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title.Contains()` code above is a [Lambda Expression](#). Lambdas are used in method-based [LINQ](#) queries as arguments to standard query operator methods such as the [Where](#) method or [Contains](#) (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see [Query Execution](#).

Note: The [Contains](#) method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, [Contains](#) maps to [SQL LIKE](#), which is case insensitive. In SQLite, with the default collation, it's case sensitive.

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Startup.cs`.

C#

Copy

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

Change the parameter to `id` and all occurrences of `searchString` change to `id`.

The previous `Index` method:

C#

Copy

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

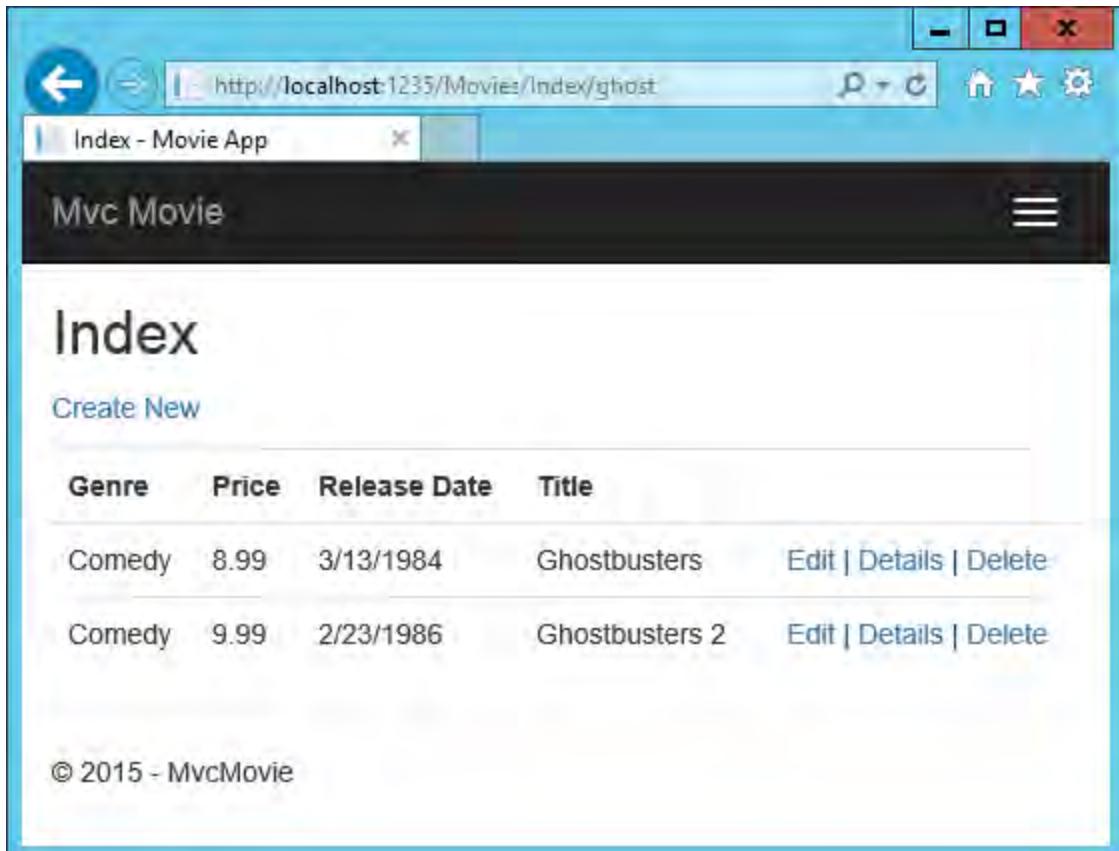
    if (!String.IsNullOrEmpty(searchString))
```

```
{  
    movies = movies.Where(s => s.Title.Contains(searchString));  
}  
  
return View(await movies.ToListAsync());  
}
```

The updated `Index` method with `id` parameter:

C#	 Copy
<pre>public async Task<IActionResult> Index(string id) { var movies = from m in _context.Movie select m; if (!String.IsNullOrEmpty(id)) { movies = movies.Where(s => s.Title.Contains(id)); } return View(await movies.ToListAsync()); }</pre>	

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```
C#  
Copy  
  
public async Task<IActionResult> Index(string searchString)  
{  
    var movies = from m in _context.Movie  
                select m;  
  
    if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
  
    return View(await movies.ToListAsync());  
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

```
CSHTML  
Copy  
  
<form action="/Movies/Index" method="get">
```

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

    <h2>Index</h2>

    <p>
        <a asp-action="Create">Create New</a>
    </p>

    <form asp-controller="Movies" asp-action="Index">
        <p>
            Title: <input type="text" name="SearchString" />
            <input type="submit" value="Filter" />
        </p>
    </form>
    <table class="table">
        <thead>
```

The HTML `<form>` tag uses the [Form Tag Helper](#), so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't

need it, because the method isn't changing the state of the app, just filtering data.

You could add the following [HttpPost] Index method.

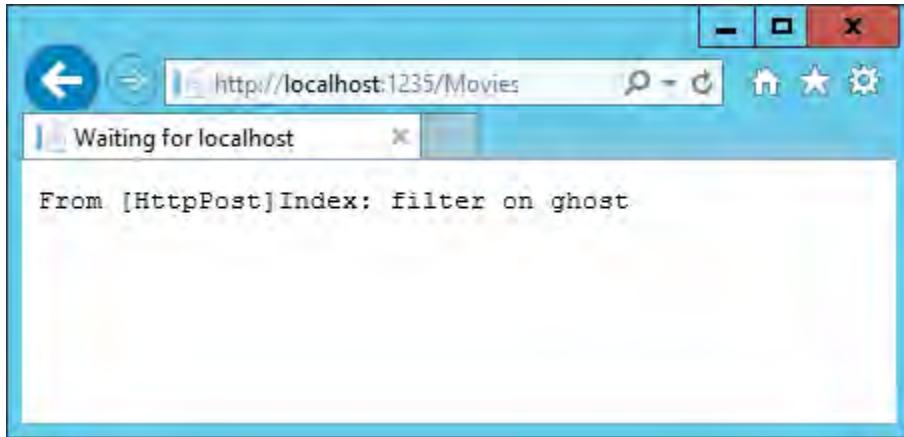
C#

 Copy

```
[HttpPost]  
public string Index(string searchString, bool notUsed)  
{  
    return "From [HttpPost]Index: filter on " + searchString;  
}
```

The notUsed parameter is used to create an overload for the Index method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the [HttpPost] Index method, and the [HttpPost] Index method would run as shown in the image below.



However, even if you add this [HttpPost] version of the Index method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (`localhost:{PORT}/Movies/Index`) -- there's no search information in the URL. The search string information is sent to the server as a form field value . You can verify that with the browser Developer tools or the excellent Fiddler tool . The image below shows the Chrome browser Developer tools:

The screenshot shows the Network tab in the Chrome DevTools Network panel. A red box highlights the request details for the URL `http://localhost:5000/Movies`. Another red box highlights the `Form Data` section, which contains the search parameter `SearchString: Ghost` and the `__RequestVerificationToken` anti-forgery token.

`Request URL: http://localhost:5000/Movies`
`Request Method: POST`
`Status Code: 200 OK`

`Form Data`

`SearchString: Ghost`
`__RequestVerificationToken: CfdJ8B98MxUFL5pAq2aeCj59HP1g2HXMD176MabW7uuk20AGreBb3y0NufBTMAjxmJCjRFe-2sF50PVla72IyfcA9Pao3muZ0f4jtjDND1XEagdJk_g67wBX12qOKi70LD980GjmjBB_-5rvRhJuQcRoPRw`

You can see the search parameter and **XSRF** token in the request body. Note, as mentioned in the previous tutorial, the **Form Tag Helper** generates an **XSRF** anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` found in the `Views/Movies/Index.cshtml` file.

CSHTML	 Copy
<pre>@model IEnumerable<MvcMovie.Models.Movie> @{ ViewData["Title"] = "Index"; } <h1>Index</h1> <p> <a asp-action="Create">Create New </p> <form asp-controller="Movies" asp-action="Index" method="get"> <p> Title: <input type="text" name="SearchString" /> <input type="submit" value="Filter" /> </p> </form> <table class="table"> <thead> <tr> <th> @Html.DisplayNameFor(model => model.Title) </th> </tr> <tbody> <tr> <td> <div> @(item.Id) </div> <div> @(item.Title) @(item.ReleaseDate) @(item.Genre) </div> </td> </tr> <tbody> </table></pre>	

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.

Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete

The following markup shows the change to the `form` tag:

CSHTML	Copy
<form asp-controller="Movies" asp-action="Index" method="get">	

Add Search by genre

Add the following `MovieGenreViewModel` class to the `Models` folder:

C#	Copy
using Microsoft.AspNetCore.Mvc.Rendering; using System.Collections.Generic; namespace MvcMovie.Models { public class MovieGenreViewModel { public List<Movie> Movies { get; set; } public SelectList Genres { get; set; } public string MovieGenre { get; set; } public string SearchString { get; set; } } }	

```
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

C#

 Copy

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync(),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a LINQ query that retrieves all the genres from the database.

C#

 Copy

```
// Use LINQ to get list of genres.  
IQueryable<string> genreQuery = from m in _context.Movie  
                                orderby m.Genre  
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in `Views/Movies/` as follows:

CSHTML

 Copy

```
@model MvcMovie.Models.MovieGenreViewModel  
  
{@  
    ViewData["Title"] = "Index";  
}  
  
<h1>Index</h1>  
  
<p>  
    <a asp-action="Create">Create New</a>  
</p>  
<form asp-controller="Movies" asp-action="Index" method="get">  
    <p>  
        <select asp-for="MovieGenre" asp-items="Model.Genres">  
            <option value="">All</option>  
        </select>  
  
        Title: <input type="text" asp-for="SearchString" />  
        <input type="submit" value="Filter" />  
    </p>  
</form>  
  
<table class="table">  
    <thead>
```

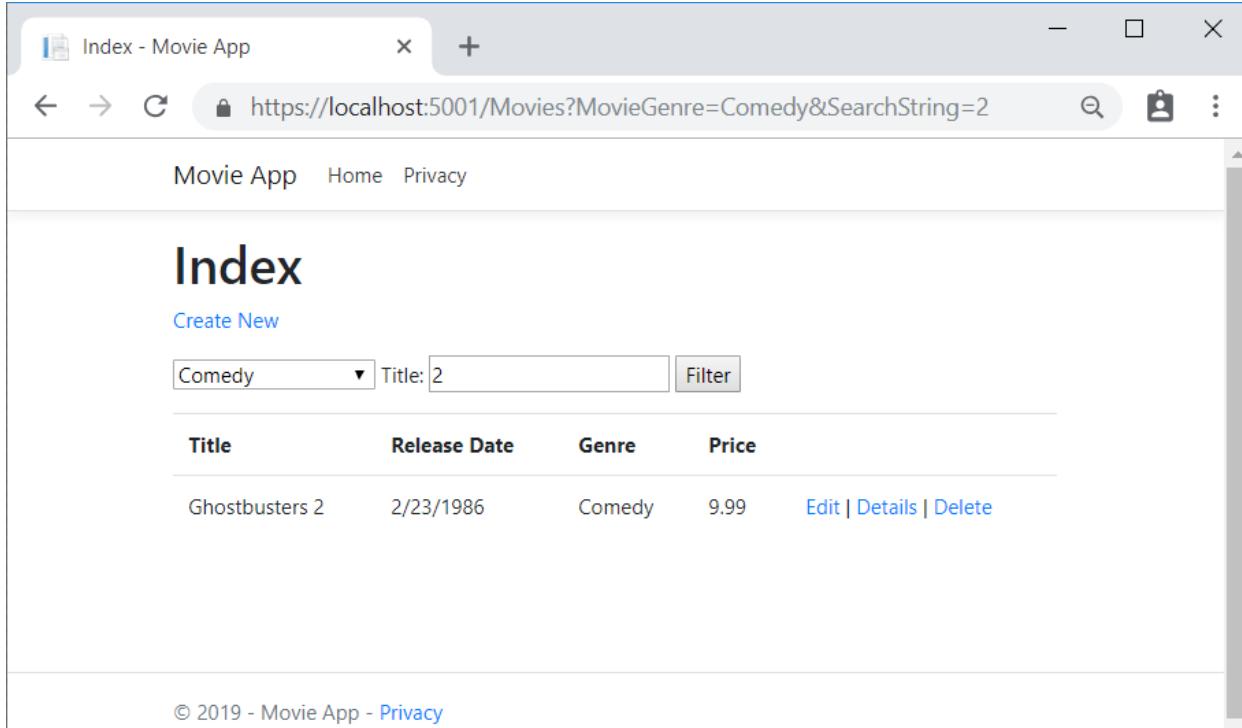
```
<tr>
    <th>
        @Html.DisplayNameFor(model => model.Movies[0].Title)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movies[0].Genre)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movies[0].Price)
    </th>
    <th></th>
</tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
        <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>
```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movies[0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both:



The screenshot shows a browser window titled "Index - Movie App". The address bar contains the URL `https://localhost:5001/Movies?MovieGenre=Comedy&SearchString=2`. The page header includes "Movie App" and navigation links for "Home" and "Privacy". The main content area has a title "Index" and a "Create New" link. Below is a search/filter section with dropdowns for "Genre" (set to "Comedy") and "Title" (set to "2"), and a "Filter" button. A table lists movies with columns: Title, Release Date, Genre, and Price. One row is shown: "Ghostbusters 2", "2/23/1986", "Comedy", "9.99", with "Edit | Details | Delete" links. At the bottom, there is a copyright notice "© 2019 - Movie App - Privacy".

Title	Release Date	Genre	Price
Ghostbusters 2	2/23/1986	Comedy	9.99

[Previous](#)[Next](#)

Part 8, add a new field to an ASP.NET Core MVC app

Article • 09/27/2021 • 14 minutes to read •  +14

[Is this page helpful?](#)

In this article

[Add a Rating Property to the Movie Model](#)

By [Rick Anderson](#)

In this section [Entity Framework](#) Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to `Models/Movie.cs`:

C#	 Copy
<pre>using System; using System.ComponentModel.DataAnnotations; using System.ComponentModel.DataAnnotations.Schema; namespace MvcMovie.Models { public class Movie { public int Id { get; set; } public string Title { get; set; }</pre>	

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }
public string Genre { get; set; }

[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
public string Rating { get; set; }
}

}
```

Build the app

Visual Studio

Visual Studio Code

Visual Studio for Mac

Ctrl+Shift+B

Because you've added a new field to the `Movie` class, you need to update the property binding list so this new property will be included. In `MoviesController.cs`, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

C#

 Copy

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

Edit the `/Views/Movies/Index.cshtml` file and add a `Rating` field:

CSHTML

 Copy

```
<thead>
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Genre)
    </th>
    <th>
```

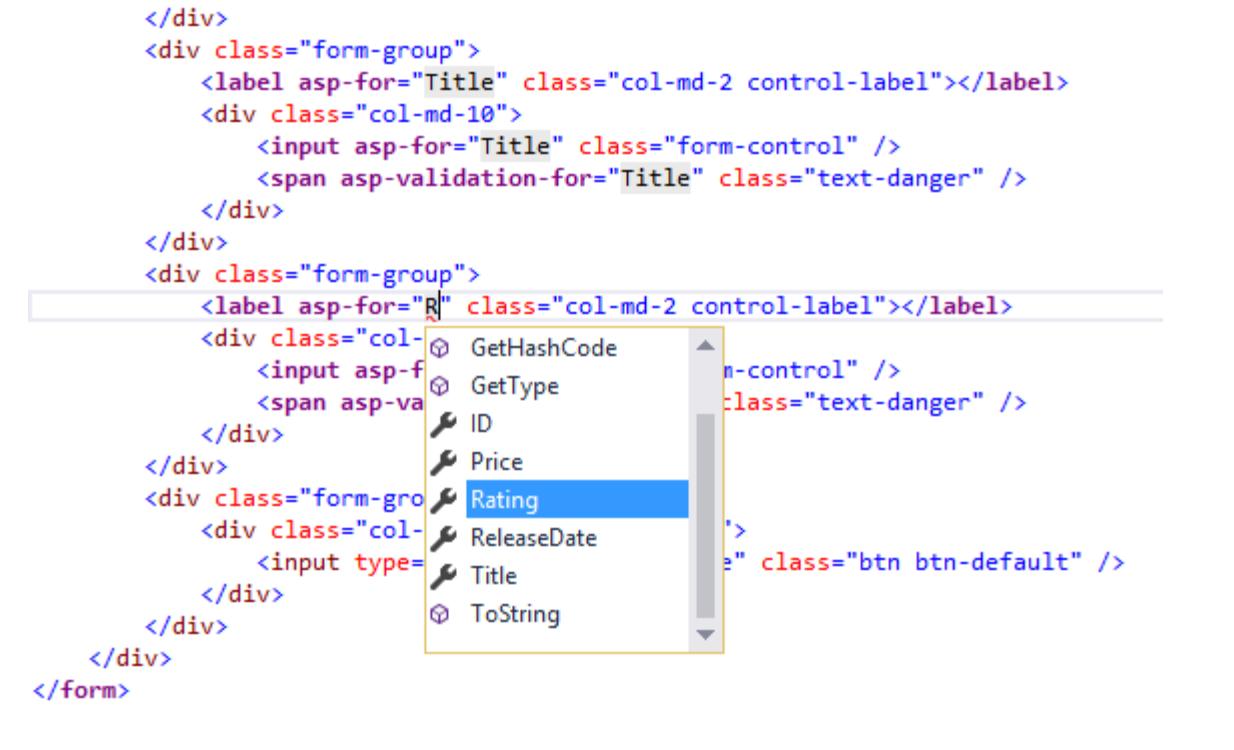
```
</th>
<th>
    @Html.DisplayNameFor(model => model.Movies[0].Price)
</th>
<th>
    @Html.DisplayNameFor(model => model.Movies[0].Rating)
</th>
<th></th>
</tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => 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>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>
```

Update the `/Views/Movies/Create.cshtml` with a `Rating` field.

Visual Studio / Visual Studio for Mac

Visual Studio Code

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with [Tag Helpers](#).



The screenshot shows a portion of an ASP.NET Core MVC view file (Index.cshtml) with code completion. The cursor is over the 'Rating' field in the 'asp-for' attribute of an input element. A dropdown menu lists several properties of the 'Movie' class, with 'Rating' highlighted in blue. Other visible properties include GetHashCode, GetType, ID, Price, ReleaseDate, Title, and ToString.

```
</div>
<div class="form-group">
    <label asp-for="Title" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger" />
    </div>
</div>
<div class="form-group">
    <label asp-for="R" class="col-md-2 control-label"></label>
    <div class="col->
```

Update the remaining templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

C#	 Copy
<pre>new Movie { Title = "When Harry Met Sally", ReleaseDate = DateTime.Parse("1989-1-11"), Genre = "Romantic Comedy", Rating = "R", Price = 7.99M },</pre>	

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

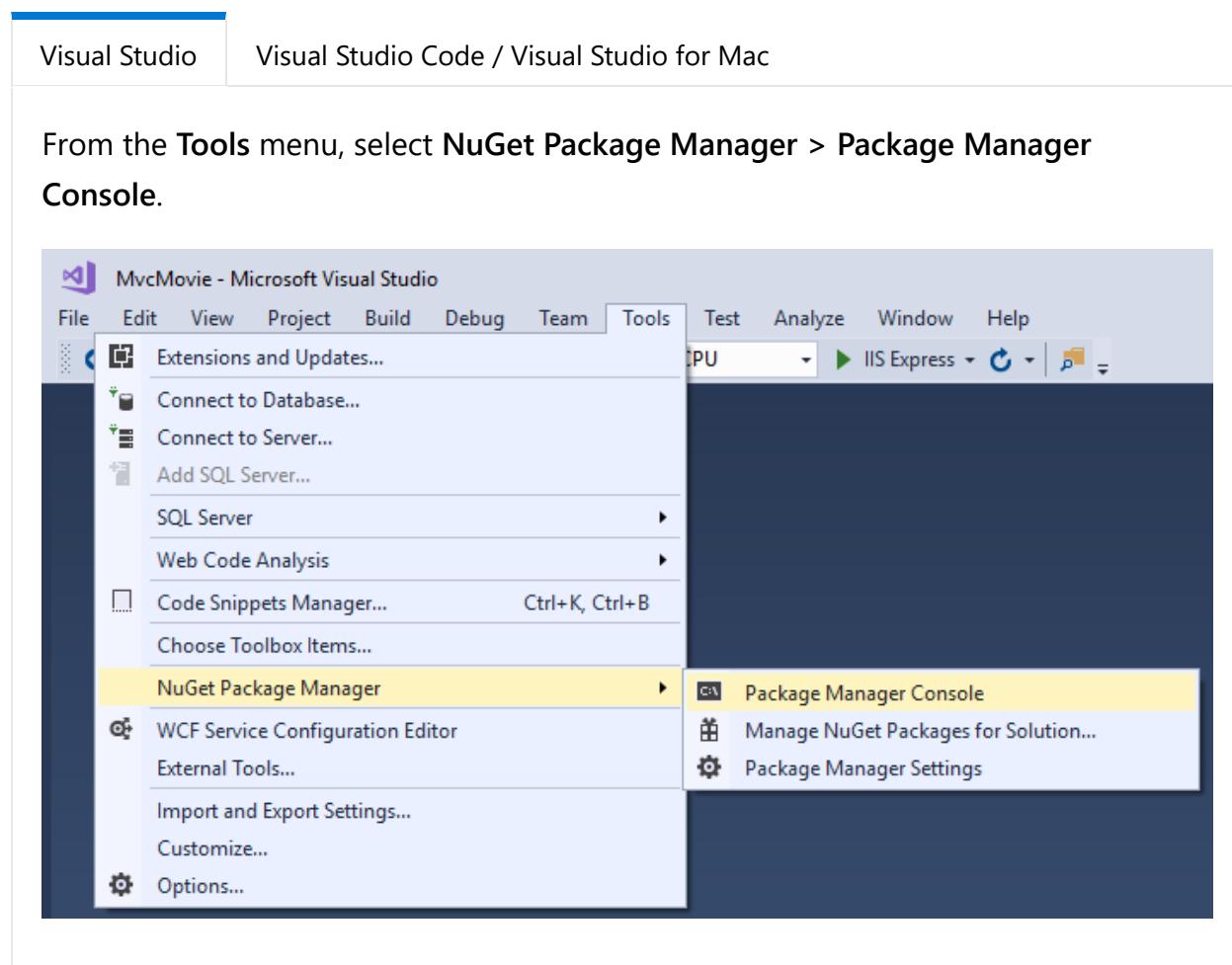
```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.



In the PMC, enter the following commands:

PowerShell

 Copy

```
Add-Migration Rating  
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

[Previous](#)

[Next](#)

Part 9, add validation to an ASP.NET Core MVC app

Article • 09/27/2021 • 19 minutes to read •  +6

[Is this page helpful?](#)

In this article

[Keeping things DRY](#)

[Add validation rules to the movie model](#)

[Validation Error UI](#)

[How validation works](#)

[Using DataType Attributes](#)

[Additional resources](#)

By [Rick Anderson](#)

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is [DRY](#) ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Add validation rules to the movie model

The `DataAnnotations` namespace provides a set of built-in validation attributes that are

applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

C#	 Copy
<pre>public class Movie { public int Id { get; set; } [StringLength(60, MinimumLength = 3)] [Required] public string Title { get; set; } [Display(Name = "Release Date")] [DataType(DataType.Date)] public DateTime ReleaseDate { get; set; } [Range(1, 100)] [DataType(DataType.Currency)] [Column(TypeName = "decimal(18, 2)")] public decimal Price { get; set; } [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*\$")] [Required] [StringLength(30)] public string Genre { get; set; } [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*\$")] [StringLength(5)] [Required] public string Rating { get; set; } }</pre>	

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":

- Must only use letters.
- The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
 - Requires that the first character be an uppercase letter.
 - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.

The screenshot shows a browser window titled "Create - Movie App" with the URL "localhost:5000/Movies/Create". The page has a dark header with the text "MvcMovie" and a three-line menu icon. The main content area has a light gray background and displays a "Create" form for a movie. The form consists of five input fields: "Title", "Release Date", "Genre", "Price", and "Rating". Each field has a red validation message below it:

- "Title": "The field Title must be a string with a minimum length of 3 and a maximum length of 60."
- "Release Date": "The Genre field is required."
- "Genre": "The field Price must be a number."
- "Price": "The field Rating must match the regular expression '^([A-Z]+[a-zA-Z]*\$)'."

Below the form is a "Create" button and a "Back to List" link. At the bottom of the page, there is a footer with the text "© 2017 - MvcMovie".

⚠ Note

You may not be able to enter decimal commas in decimal fields. To support **jQuery validation** for non-English locales that use a comma (",") for a decimal point, and

non US-English date formats, you must take steps to globalize your app. See this [GitHub issue 4076](#) for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the [Fiddler tool](#), or the [F12 Developer tools](#).

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

C#

 Copy

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

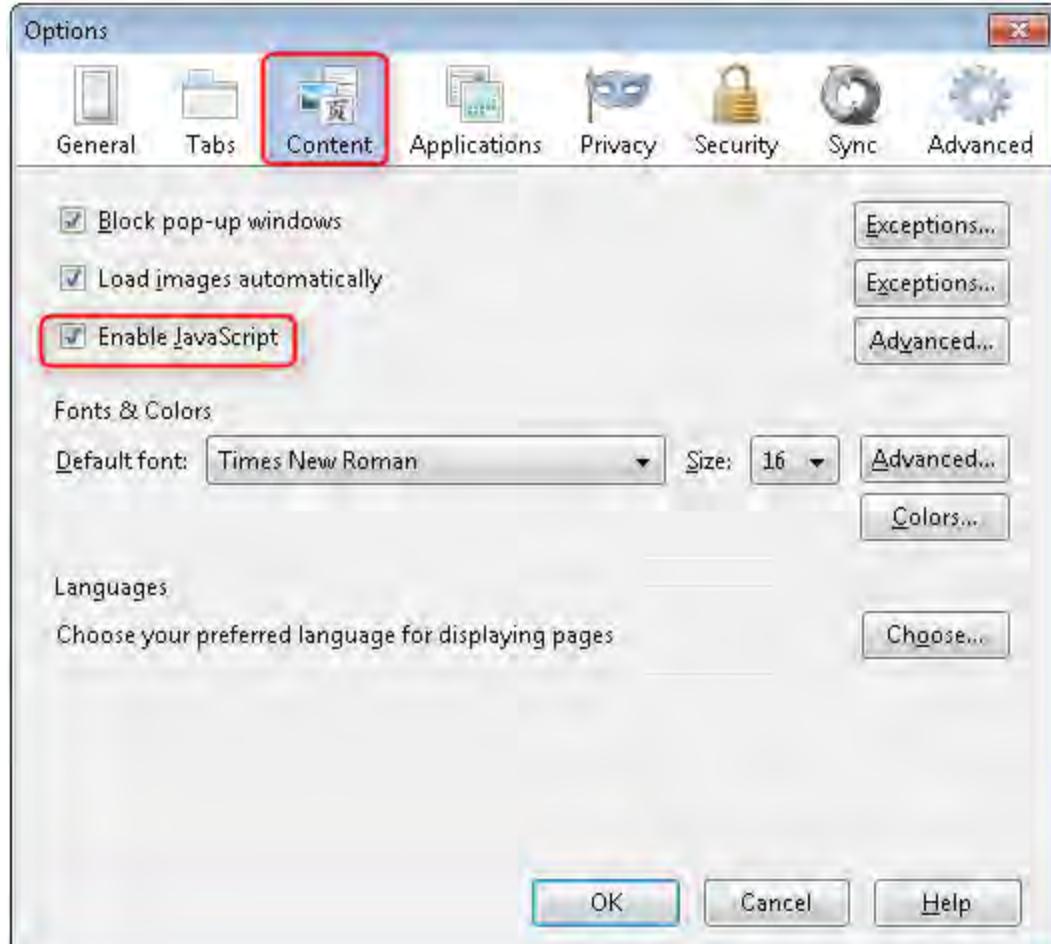
// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
```

```
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

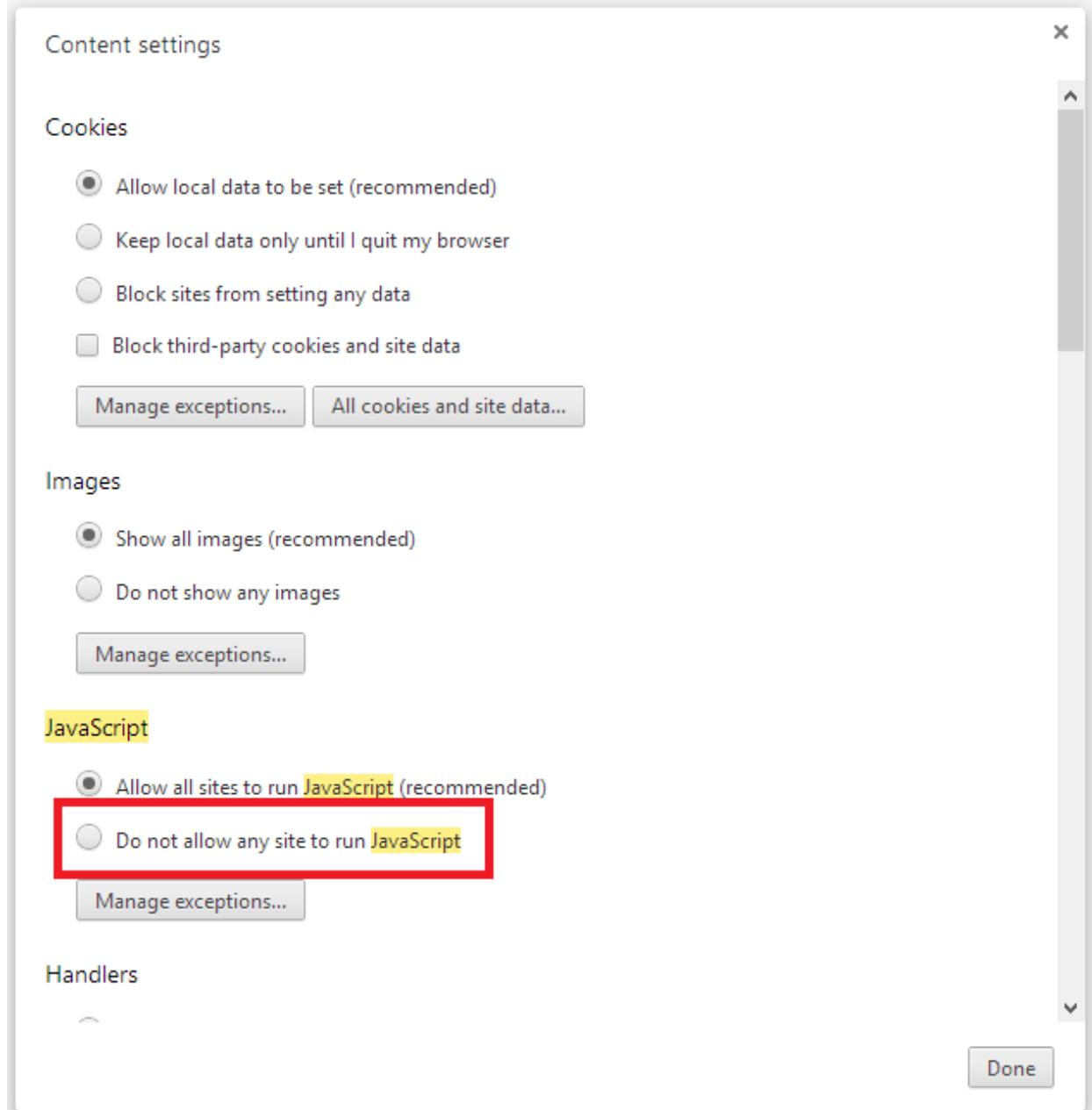
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost]` `Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

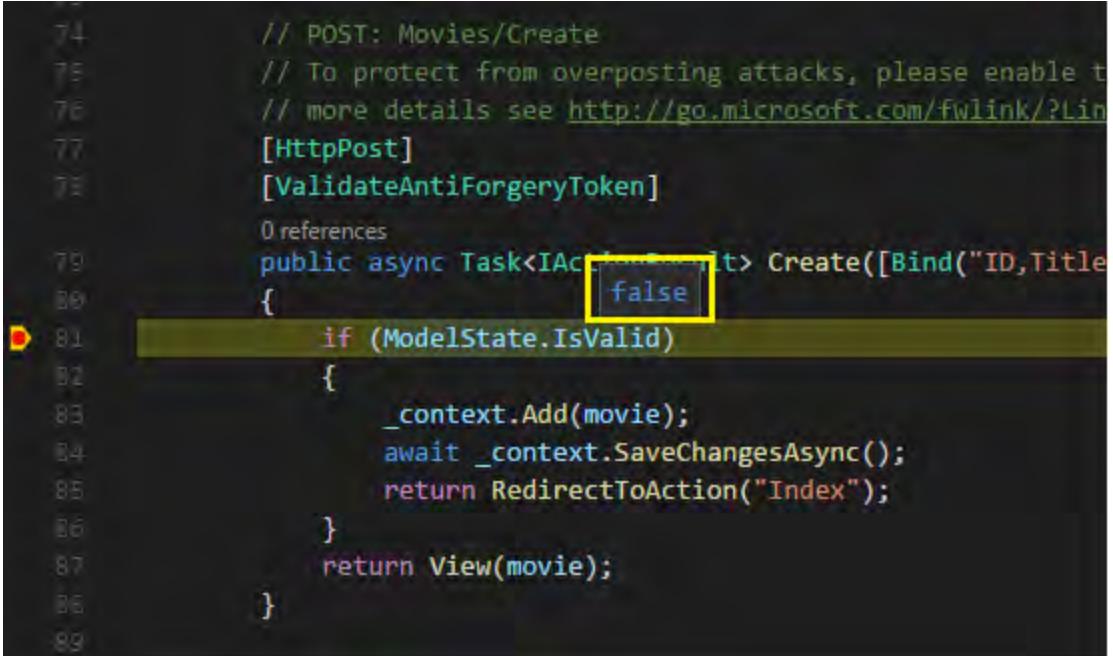
The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.



```
74     // POST: Movies/Create
75     // To protect from overposting attacks, please enable t
76     // more details see http://go.microsoft.com/fwlink/?LinkID=208113
77     [HttpPost]
78     [ValidateAntiForgeryToken]
79     public async Task Create([Bind("ID,Title")]
80     {
81         if (ModelState.IsValid)
82         {
83             _context.Add(movie);
84             await _context.SaveChangesAsync();
85             return RedirectToAction("Index");
86         }
87         return View(movie);
88     }
89 }
```

The portion of the *Create.cshtml* view template is shown in the following markup:

HTML	 Copy
<pre><h4>Movie</h4> <hr /> <div class="row"> <div class="col-md-4"> <form asp-action="Create"> <div asp-validation-summary="ModelOnly" class="text-danger"> </div> <div class="form-group"> <label asp-for="Title" class="control-label"></label> <input asp-for="Title" class="form-control" /> </div> <!-- Markup removed for brevity. --> </div> </div></pre>	

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The [Input Tag Helper](#) uses the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](#) displays validation errors. See [Validation](#) for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views/templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using `DataType` Attributes

Open the `Movie.cs` file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

C#

 Copy

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies elements/attributes such as `<a>` for URL's and `` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`,

EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do not provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

C#

 Copy

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

 **Note**

jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

C#

 Copy

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^([A-Z][a-zA-Z\s]*$"), Required, StringLength(30))]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^([A-Z][a-zA-Z0-9]*'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- [Working with Forms](#)
- [Globalization and localization](#)

- [Introduction to Tag Helpers](#)
- [Author Tag Helpers](#)

[Previous](#)[Next](#)

Part 10, examine the Details and Delete methods of an ASP.NET Core app

Article • 09/27/2021 • 6 minutes to read •  +9

[Is this page helpful?](#)

By [Rick Anderson](#)

Open the Movie controller and examine the `Details` method:

C#	 Copy
<pre>// 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); }</pre>	

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in `Startup.cs`.

C#	 Copy
<pre>app.UseEndpoints(endpoints => { endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");</pre>	

```
});
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

C#

 Copy

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(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);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a

view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The [HttpPost] method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

C#

 Copy

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

C#

 Copy

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed`

parameter. You could do the same thing here for the [HttpPost] Delete method:

C#

 Copy

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see [Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service](#).

[Previous](#)

Introduction to ASP.NET Core

Article • 09/22/2021 • 9 minutes to read •  +3

[Is this page helpful?](#)

In this article

- [Why choose ASP.NET Core?](#)
- [Build web APIs and web UI using ASP.NET Core MVC](#)
- [Client-side development](#)
- [ASP.NET Core target frameworks](#)
- [Recommended learning path](#)
- [Migrate from .NET Framework](#)
- [How to download a sample](#)
- [Breaking changes and security advisories](#)
- [Next steps](#)

By [Daniel Roth](#) , [Rick Anderson](#) , and [Shaun Luttin](#)

ASP.NET Core is a cross-platform, high-performance, [open-source](#) framework for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

- Build web apps and services, [Internet of Things \(IoT\)](#) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on [.NET Core](#).

Why choose ASP.NET Core?

Millions of developers use or have used [ASP.NET 4.x](#) to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, including architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- [Razor Pages](#) makes coding page-focused scenarios easier and more productive.

- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused .
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance , and modular HTTP request pipeline.
- Ability to host on the following:
 - Kestrel
 - IIS
 - HTTP.sys
 - Nginx
 - Apache
 - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including [Blazor](#), [Angular](#), [React](#), and [Bootstrap](#). For more information, see [Introduction to ASP.NET Core Blazor](#) and related topics under *Client-side development*.

ASP.NET Core target frameworks

ASP.NET Core 3.x and later can only target .NET Core. Generally, ASP.NET Core is composed of [.NET Standard](#) libraries. Libraries written with .NET Standard 2.0 run on any [.NET platform that implements .NET Standard 2.0](#).

There are several advantages to targeting .NET Core, and these advantages increase with each release. Some advantages of .NET Core over .NET Framework include:

- Cross-platform. Runs on Windows, macOS, and Linux.
- Improved performance
- [Side-by-side versioning](#)
- New APIs
- Open source

Recommended learning path

We recommend the following sequence of tutorials for an introduction to developing ASP.NET Core apps:

1. Follow a tutorial for the app type you want to develop or maintain.

App type	Scenario	Tutorial
Web app	New server-side web UI development	Get started with Razor Pages
Web app	Maintaining an MVC app	Get started with MVC
Web app	Client-side web UI development	Get started with Blazor

App type	Scenario	Tutorial
Web API	RESTful HTTP services	Create a web API†
Remote Procedure Call app	Contract-first services using Protocol Buffers	Get started with a gRPC service
Real-time app	Bidirectional communication between servers and connected clients	Get started with SignalR

2. Follow a tutorial that shows how to do basic data access.

Scenario	Tutorial
New development	Razor Pages with Entity Framework Core
Maintaining an MVC app	MVC with Entity Framework Core

3. Read an overview of ASP.NET Core [fundamentals](#) that apply to all app types.

4. Browse the table of contents for other topics of interest.

†There's also an [interactive web API tutorial](#). No local installation of development tools is required. The code runs in an [Azure Cloud Shell](#) in your browser, and [curl](#) is used for testing.

Migrate from .NET Framework

For a reference guide to migrating ASP.NET 4.x apps to ASP.NET Core, see [Migrate from ASP.NET to ASP.NET Core](#).

How to download a sample

Many of the articles and tutorials include links to sample code.

1. [Download the ASP.NET repository zip file](#).
2. Unzip the `AspNetCore.Docs-main.zip` file.
3. To access an article's sample app in the unzipped repository, use the URL in the article's sample link to help you navigate to the sample's folder. Usually, an article's

sample link appears at the top of the article with the link text *View or download sample code*.

Preprocessor directives in sample code

To demonstrate multiple scenarios, sample apps use the `#define` and `#if-#else/#elif-#endif` preprocessor directives to selectively compile and run different sections of sample code. For those samples that make use of this approach, set the `#define` directive at the top of the C# files to define the symbol associated with the scenario that you want to run. Some samples require defining the symbol at the top of multiple files in order to run a scenario.

For example, the following `#define` symbol list indicates that four scenarios are available (one scenario per symbol). The current sample configuration runs the `TemplateCode` scenario:

C#	 Copy
<pre>#define TemplateCode // or LogFromMain or ExpandDefault or FilterInCode</pre>	

To change the sample to run the `ExpandDefault` scenario, define the `ExpandDefault` symbol and leave the remaining symbols commented-out:

C#	 Copy
<pre>#define ExpandDefault // TemplateCode or LogFromMain or FilterInCode</pre>	

For more information on using [C# preprocessor directives](#) to selectively compile sections of code, see [#define \(C# Reference\)](#) and [#if \(C# Reference\)](#).

Regions in sample code

Some sample apps contain sections of code surrounded by `#region` and `#endregion` C# directives. The documentation build system injects these regions into the rendered documentation topics.

Region names usually contain the word "snippet." The following example shows a region named `snippet_WebHostDefaults`:

C#

 Copy

```
#region snippet_WebHostDefaults
Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.UseStartup<Startup>();
});
#endregion
```

The preceding C# code snippet is referenced in the topic's markdown file with the following line:

Markdown

 Copy

```
[!code-csharp[](sample/SampleApp/Program.cs?name=snippet_WebHostDefaults)]
```

You may safely ignore (or remove) the `#region` and `#endregion` directives that surround the code. Don't alter the code within these directives if you plan to run the sample scenarios described in the topic. Feel free to alter the code when experimenting with other scenarios.

For more information, see [Contribute to the ASP.NET documentation: Code snippets](#).

Breaking changes and security advisories

Breaking changes and security advisories are reported on the [Announcements repo](#). Announcements can be limited to a specific version by selecting a **Label** filter.

Next steps

For more information, see the following resources:

- [Get started with ASP.NET Core](#)
- [Publish an ASP.NET Core app to Azure with Visual Studio](#)
- [ASP.NET Core fundamentals](#)
- [The weekly ASP.NET community standup](#) covers the team's progress and plans. It features new blogs and third-party software.

Recommended content

[Get started with ASP.NET Core](#)

A short tutorial that creates and runs a basic Hello World app using ASP.NET Core.

[Choose between ASP.NET 4.x and ASP.NET Core](#)

Explains ASP.NET Core vs. ASP.NET 4.x and how to choose between them.

[Choose an ASP.NET Core UI](#)

Understand when to use the various ASP.NET Core web UI technologies Microsoft provides and supports.

[Overview of ASP.NET Core MVC](#)

Learn how ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

[Show more ▾](#)

ASP.NET MVC 5

MVC Architecture

In this section, you will get an overview of MVC architecture. The MVC architectural pattern has existed for a long time in software engineering. All most all the languages use MVC with slight variation, but conceptually it remains the same.

Let's understand the MVC architecture in ASP.NET.

MVC stands for Model, View and Controller. MVC separates application into three components - Model, View and Controller.

Model: Model represents shape of the data and business logic. It maintains the data of the application. Model objects retrieve and store model state in a database.

Model is a data and business logic.

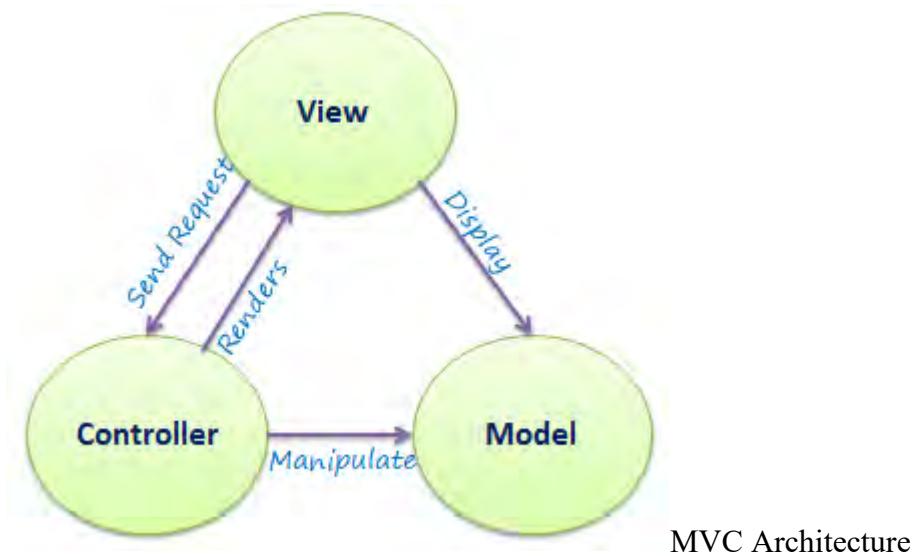
View: View is a user interface. View display data using model to the user and also enables them to modify the data.

View is a User Interface.

Controller: Controller handles the user request. Typically, user interact with View, which in-turn raises appropriate URL request, this request will be handled by a controller. The controller renders the appropriate view with the model data as a response.

Controller is a request handler.

The following figure illustrates the interaction between Model, View and Controller.



MVC Architecture

The following figure illustrates the flow of the user's request in ASP.NET MVC.



Request/Response in MVC Architecture

As per the above figure, when the user enters a URL in the browser, it goes to the server and calls appropriate controller. Then, the Controller uses the appropriate View and Model and creates the response and sends it back to the user

Create First ASP.NET MVC Application

In this section, we will create a new MVC 5 application with Visual Studio 2013 for Web and understand the basic building blocks of a MVC Application.

First of all, setup a development environment to develop an ASP.NET MVC 5 application.

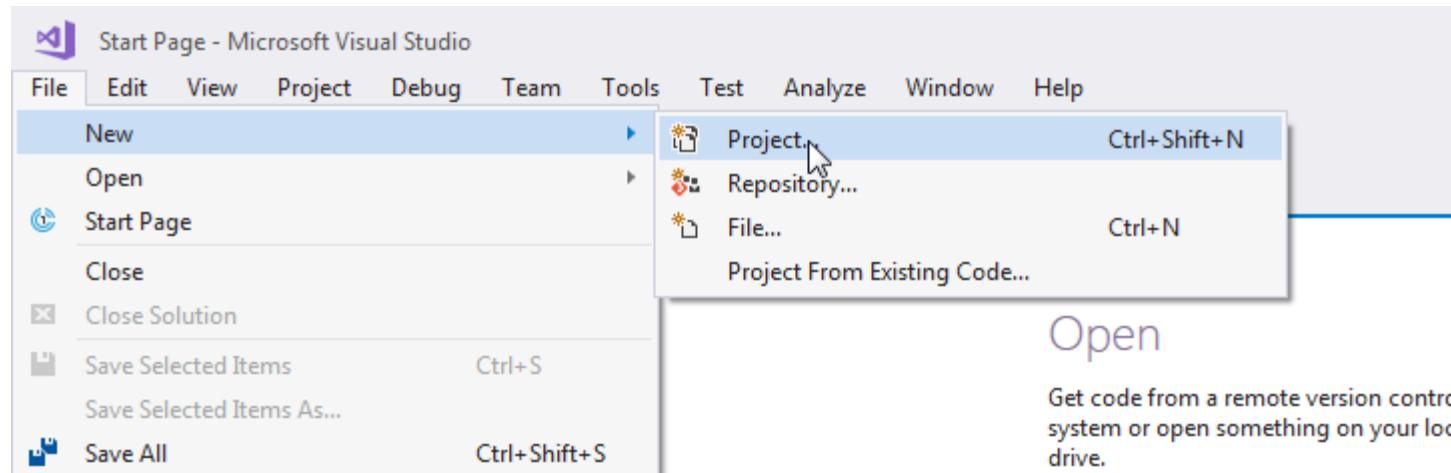
Setup Development Environment

You can develop ASP.NET MVC application with appropriate version of Visual Studio and .NET framework, as you have seen in the previous section of version history.

Here, we will use MVC v5.2, Visual Studio 2017 Community edition and .NET framework 4.6 to create our first MVC application.

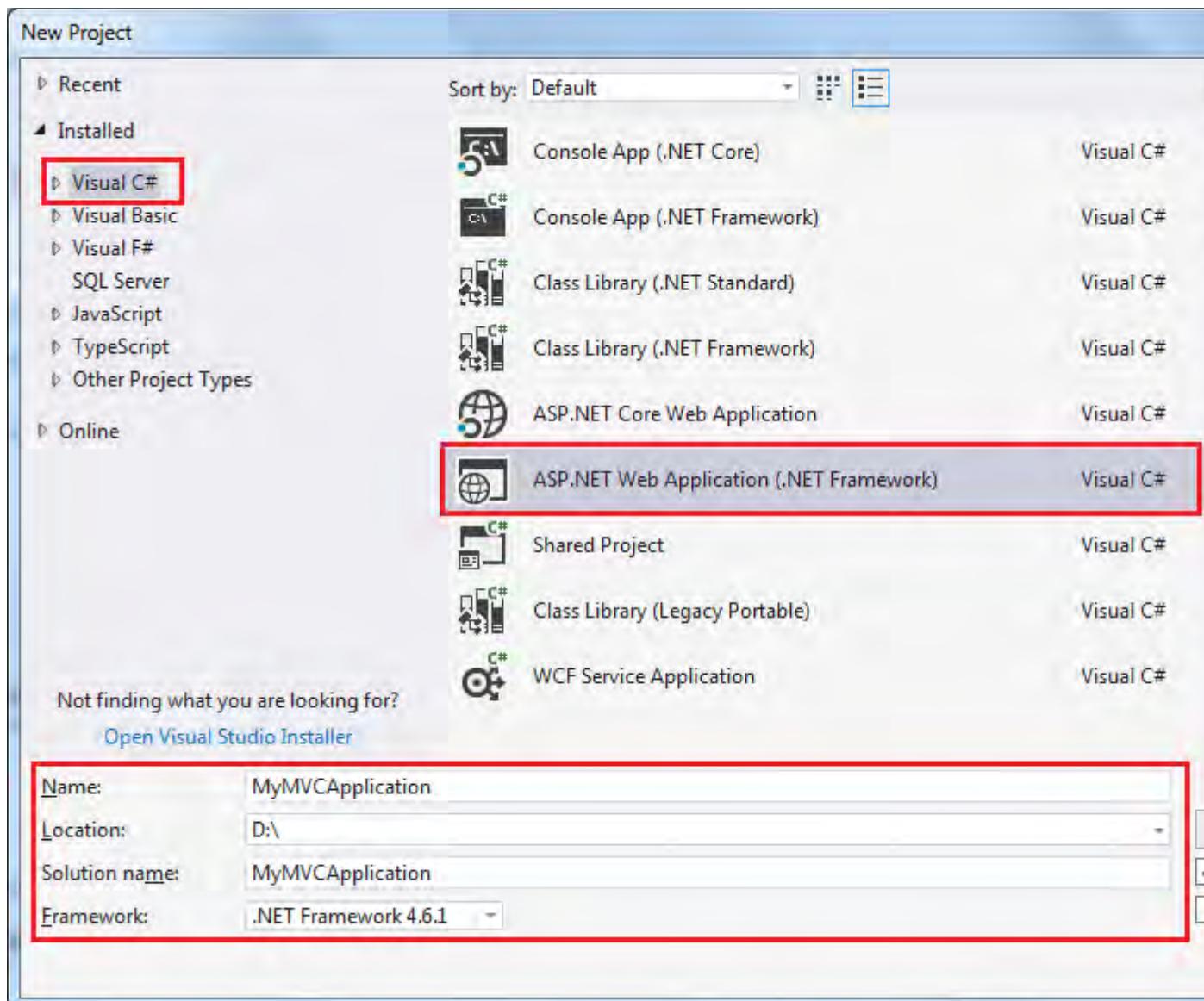
Create first simple MVC application

First of all, open a Visual Studio 2017 Community edition and select **File menu** -> **New**-> **Project** as shown below.



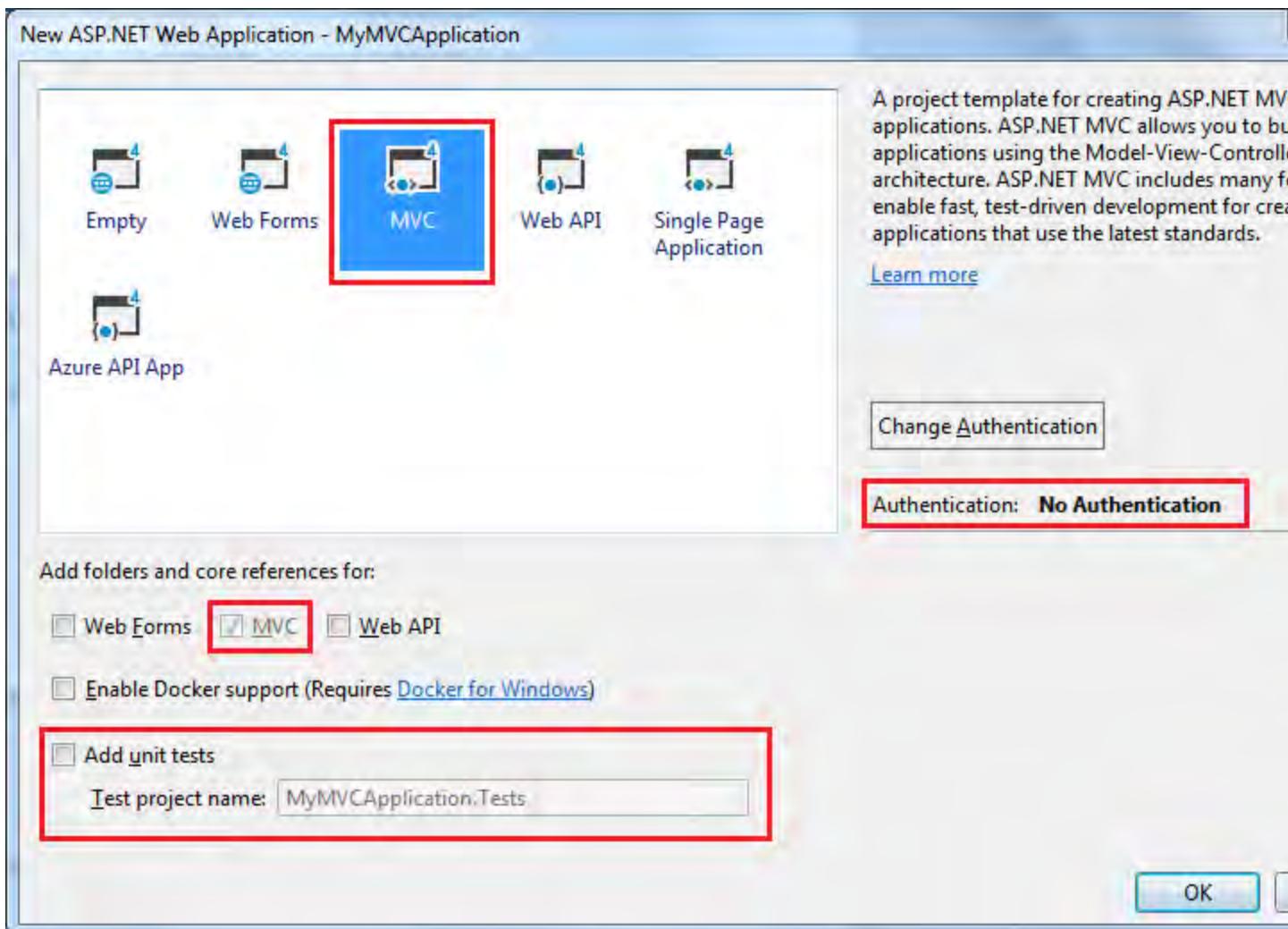
Create a New Project in Visual Studio

From the **New Project** dialog as shown below, expand Visual C# node and select **Web** in the left pane, and then select **ASP.NET Web Application (.NET Framework)** in the middle pane. Enter the name of your project **MyMVCAApplication**. (You can give any appropriate name for your application). Also, you can change the location of the MVC application by clicking on **Browse..** button. Finally, click **OK**.



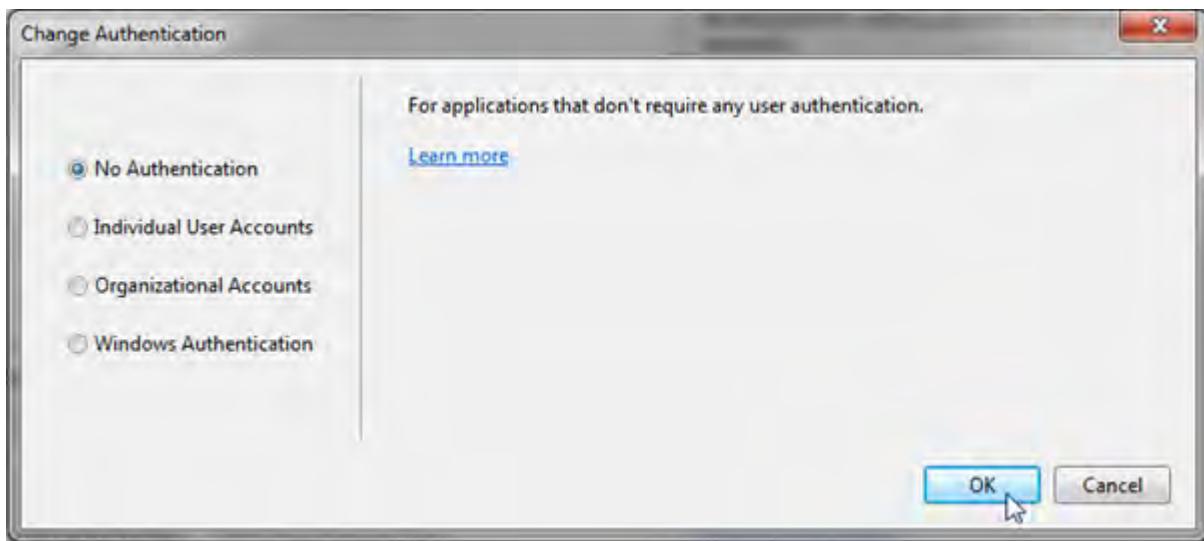
Create MVC Project in Visual Studio

From the **New ASP.NET Web Application** dialog, select MVC (if not selected already) as shown below.



Create MVC Application

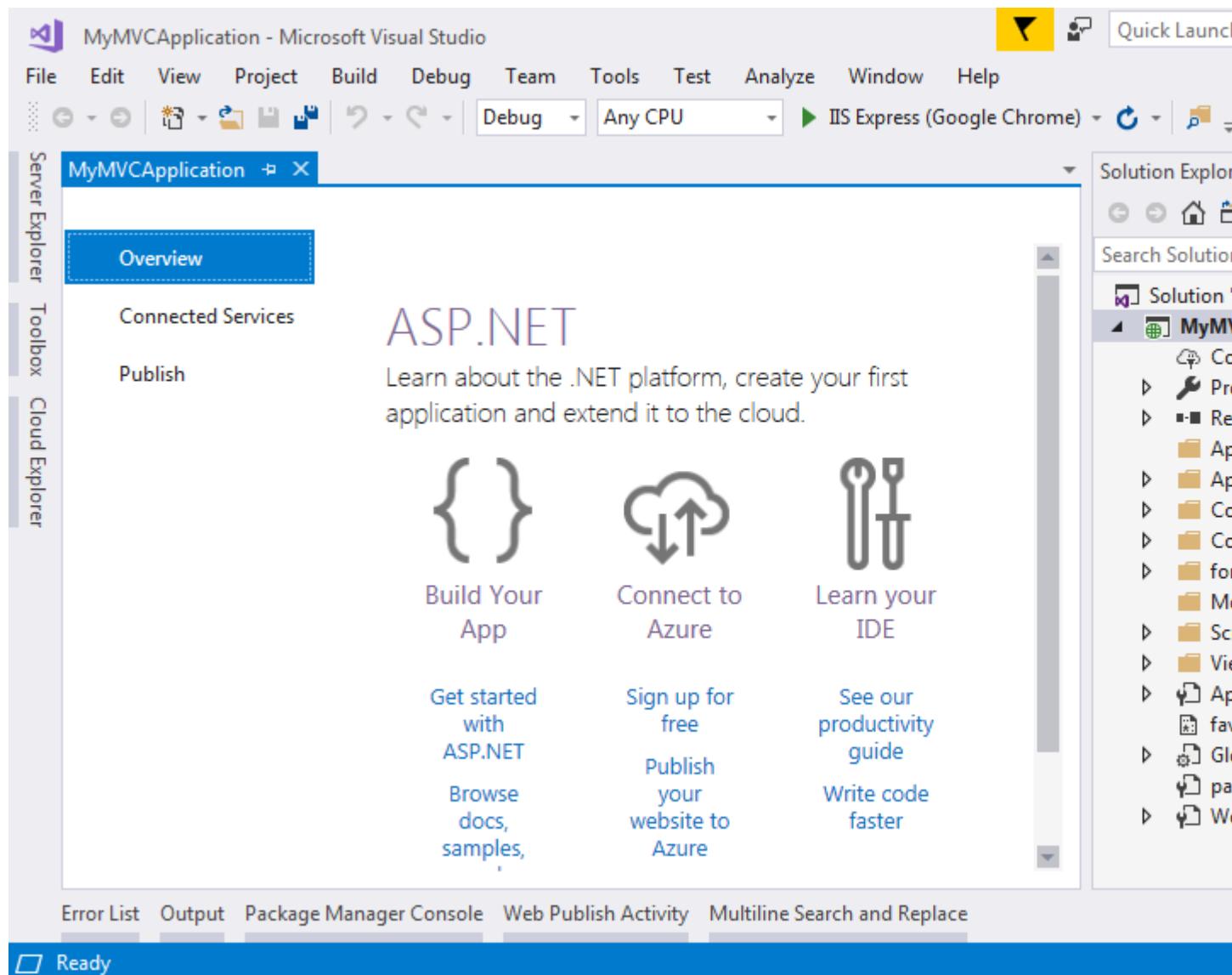
You can also change the authentication by clicking on **Change Authentication** button. You can select appropriate authentication mode for your application as shown below.



Select Authentication Type

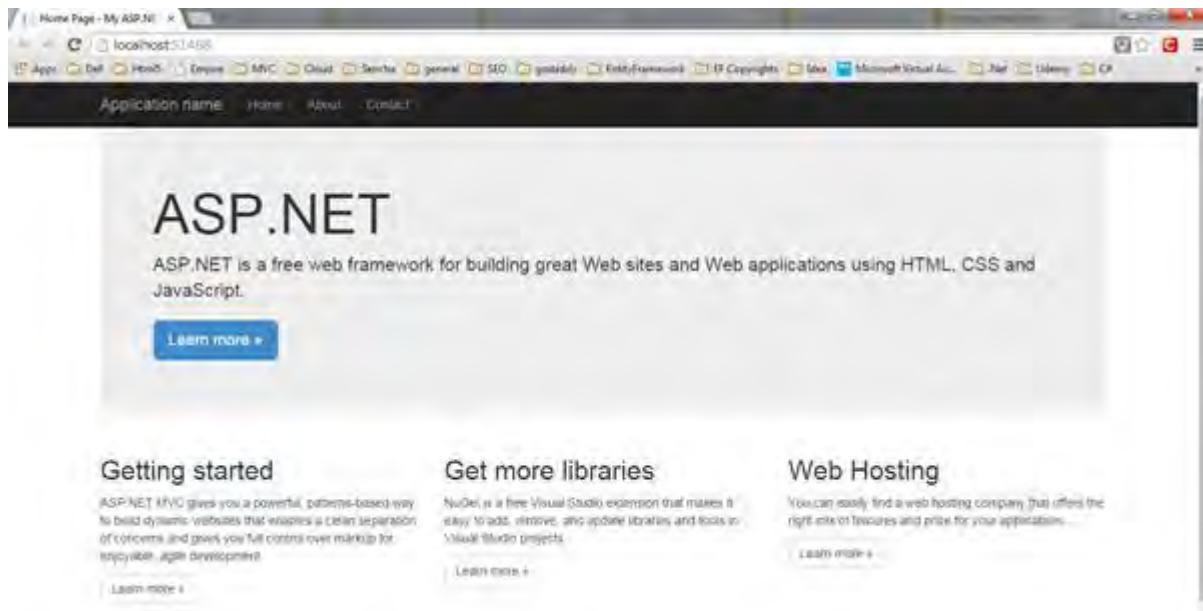
Here, we are keeping the default authentication for our application which is No Authentication. Click **OK** to continue.

Wait for some time till Visual Studio creates a simple MVC project using default template as shown below.

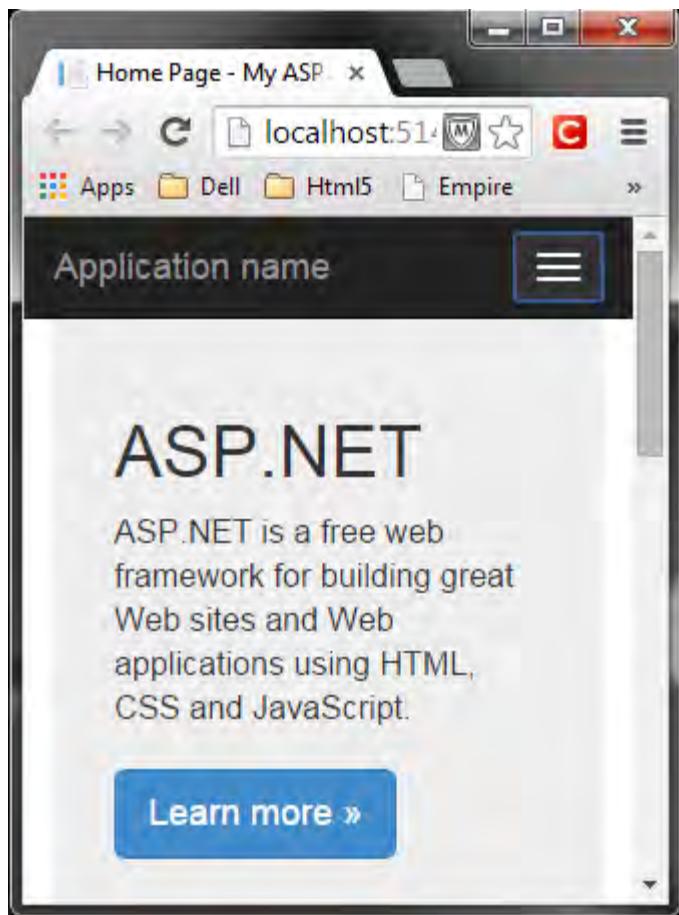


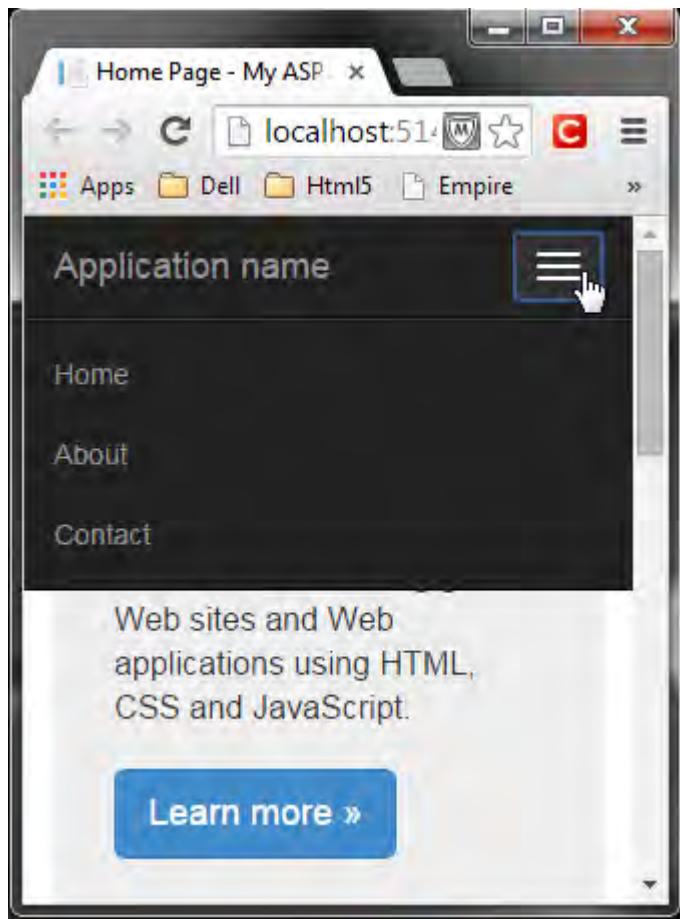
First MVC Application

Now, press F5 to run the project in debug mode or Ctrl + F5 to run the project without debugging. It will open home page in the browser as shown below.



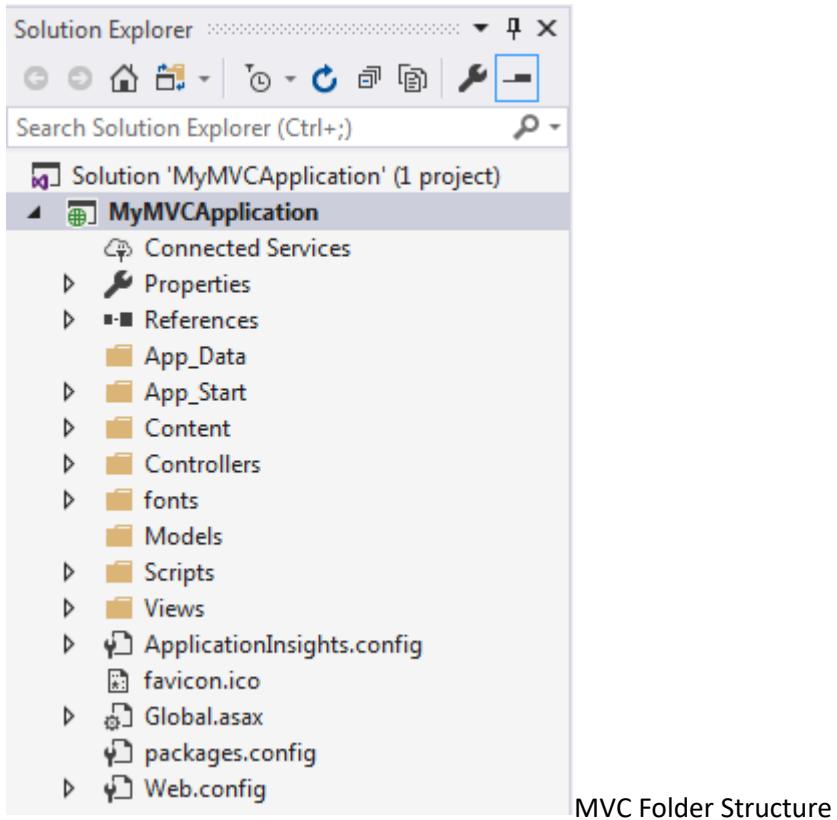
MVC 5 project includes JavaScript and CSS files of bootstrap 3.0 by default. So you can create responsive web pages. This responsive UI will change its look and feel based on the screen size of the different devices. For example, top menu bar will be changed in the mobile devices as shown below.





ASP.NET MVC Folder Structure

We have created our first MVC 5 application in the previous section. Visual Studio creates the following folder structure for MVC application by default.



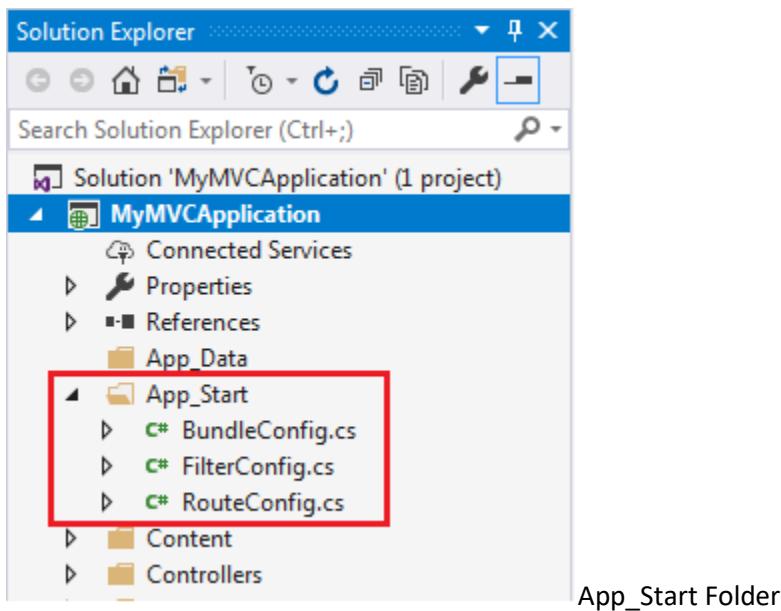
Let's see significance of each folder.

App_Data

App_Data folder can contain application data files like LocalDB, .mdf files, xml files and other data related files. IIS will never serve files from App_Data folder.

App_Start

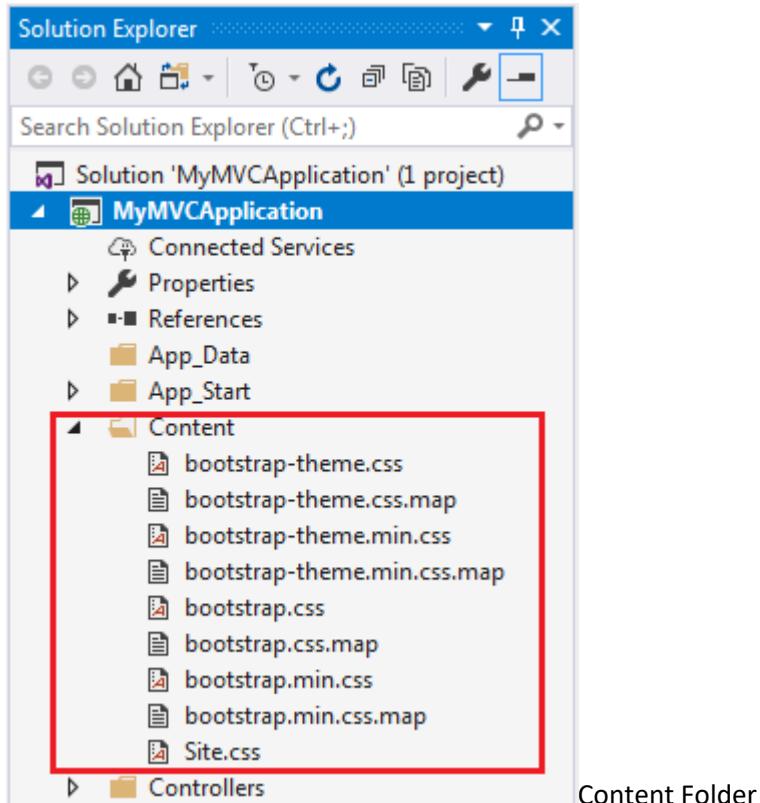
App_Start folder can contain class files which will be executed when the application starts. Typically, these would be config files like AuthConfig.cs, BundleConfig.cs, FilterConfig.cs, RouteConfig.cs etc. MVC 5 includes BundleConfig.cs, FilterConfig.cs and RouteConfig.cs by default. We will see significance of these files later.



App_Start Folder

Content

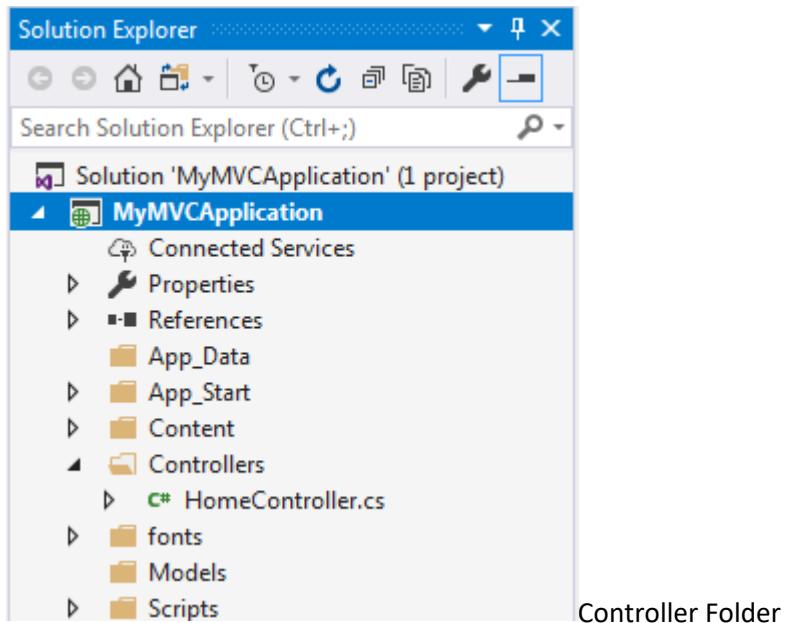
Content folder contains static files like css files, images and icons files. MVC 5 application includes bootstrap.css, bootstrap.min.css and Site.css by default.



Content Folder

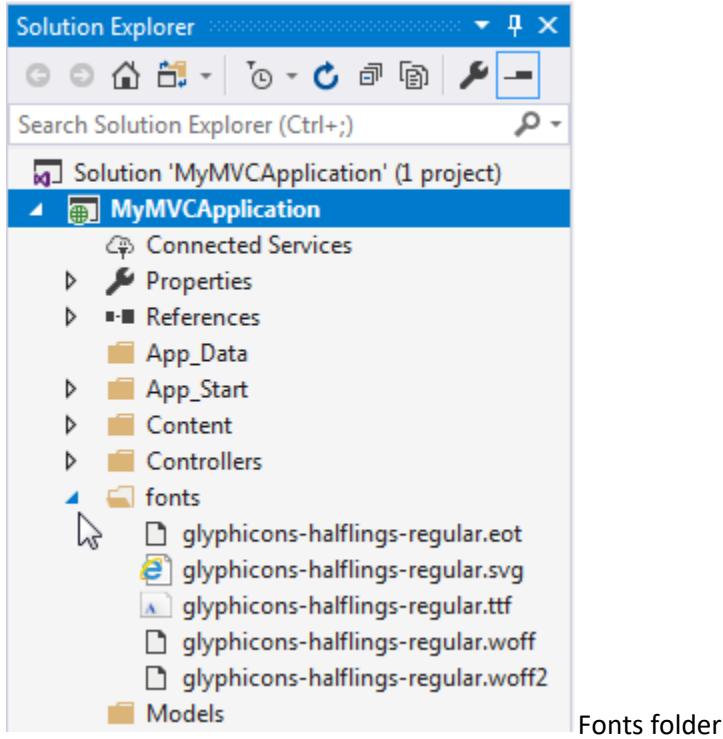
Controllers

Controllers folder contains class files for the controllers. Controllers handles users' request and returns a response. MVC requires the name of all controller files to end with "Controller". You will learn about the controller in the next section.



fonts

Fonts folder contains custom font files for your application.

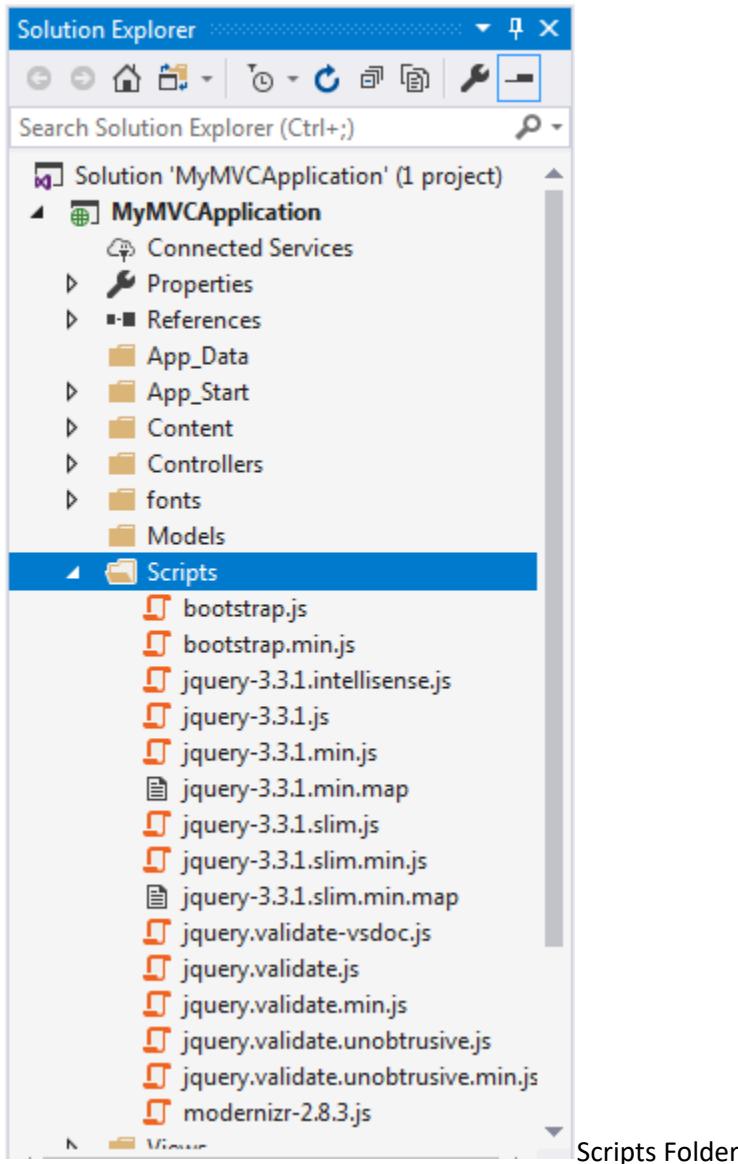


Models

Models folder contains model class files. Typically model class includes public properties, which will be used by application to hold and manipulate application data.

Scripts

Scripts folder contains JavaScript or VBScript files for the application. MVC 5 includes javascript files for bootstrap, jquery 1.10 and modernizer by default.

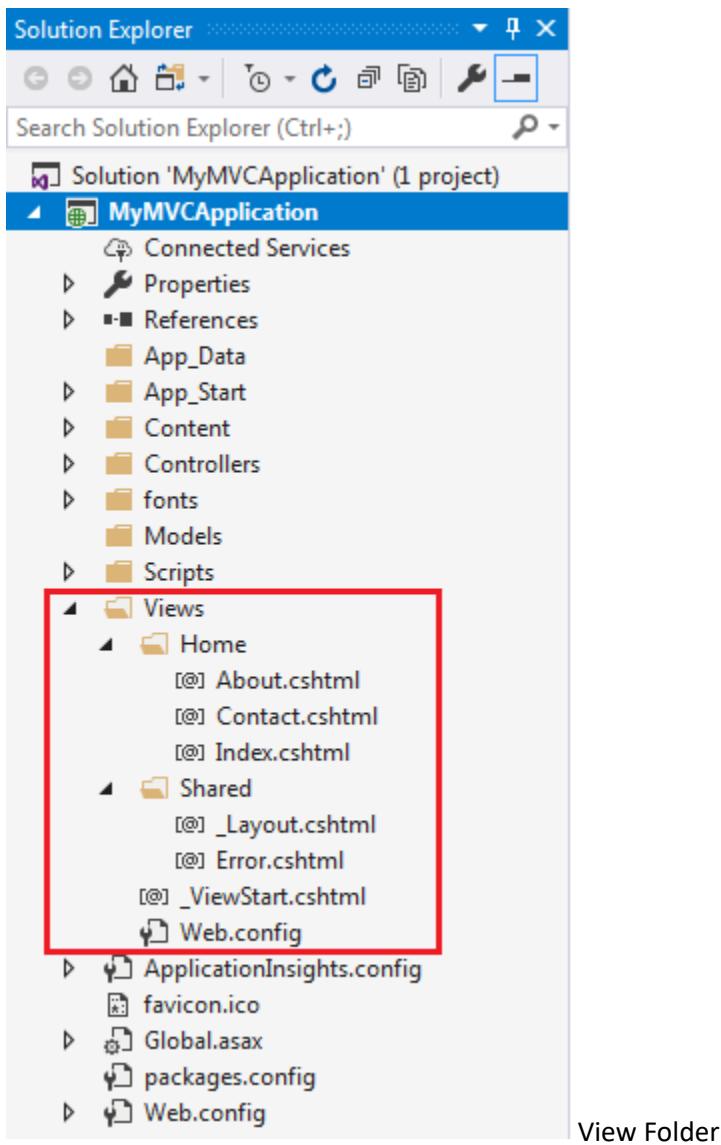


Views

Views folder contains html files for the application. Typically view file is a .cshtml file where you write html and C# or VB.NET code.

Views folder includes separate folder for each controllers. For example, all the .cshtml files, which will be rendered by HomeController will be in View > Home folder.

Shared folder under View folder contains all the views which will be shared among different controllers e.g. layout files.



Additionally, MVC project also includes following configuration files:

Global.asax

Global.asax allows you to write code that runs in response to application level events, such as Application_BeginRequest, application_start, application_error, session_start, session_end etc.

Packages.config

Packages.config file is managed by NuGet to keep track of what packages and versions you have installed in the application.

Web.config

Web.config file contains application level configurations.

Routing in MVC

In the ASP.NET Web Forms application, every URL must match with a specific .aspx file. For example, a URL `http://domain/studentsinfo.aspx` must match with the file `studentsinfo.aspx` that contains code and markup for rendering a response to the browser.

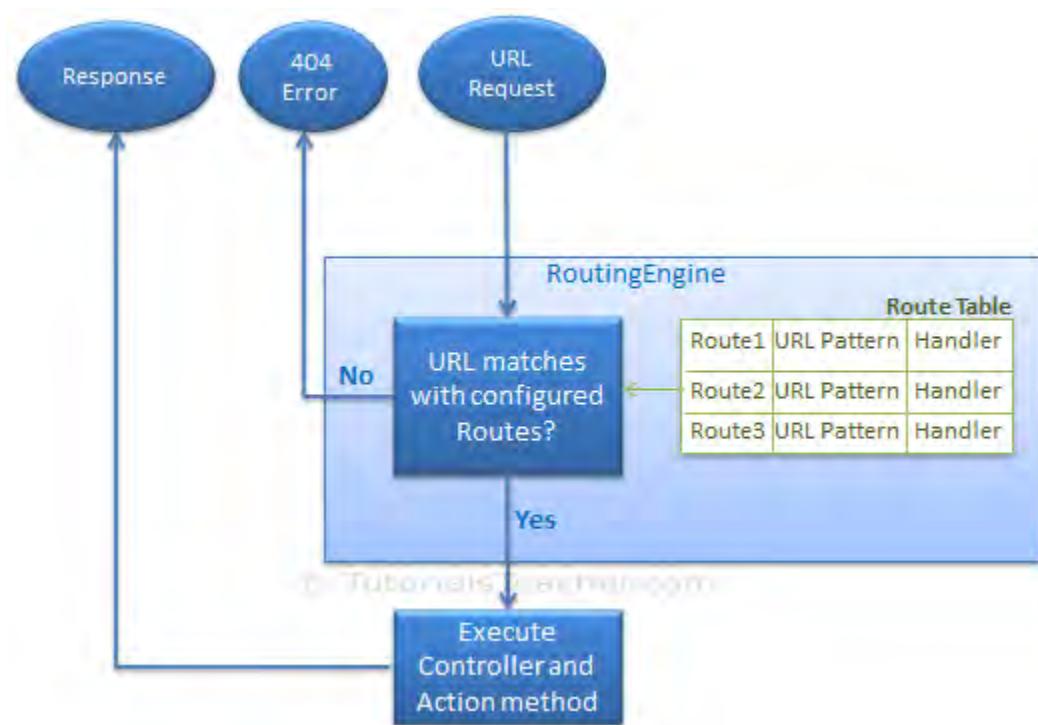
Routing is not specific to MVC framework. It can be used with ASP.NET Webform application or MVC application.

ASP.NET introduced Routing to eliminate needs of mapping each URL with a physical file. Routing enable us to define URL pattern that maps to the request handler. This request handler can be a file or class. In ASP.NET Webform application, request handler is .aspx file and in MVC, it is Controller class and Action method. For example, `http://domain/students` can be mapped to `http://domain/studentsinfo.aspx` in ASP.NET Webforms and the same URL can be mapped to Student Controller and Index action method in MVC.

Route

Route defines the URL pattern and handler information. All the configured routes of an application stored in `RouteTable` and will be used by Routing engine to determine appropriate handler class or file for an incoming request.

The following figure illustrates the Routing process.



Configure a Route

Every MVC application must configure (register) at least one route, which is configured by MVC framework by default. You can register a route in **RouteConfig** class, which is in RouteConfig.cs under **App_Start** folder. The following figure illustrates how to configure a Route in the RouteConfig class .

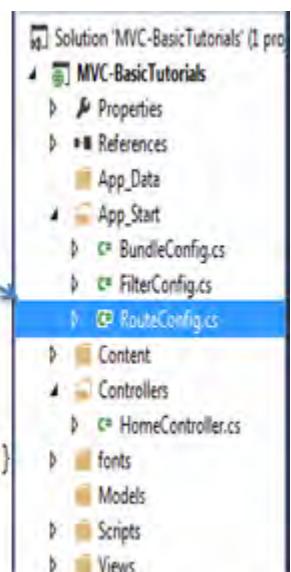
```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}"); // Route to ignore

        routes.MapRoute( // Route name
            name: "Default", // URL Pattern
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}

```

The code shows the configuration of a route named "Default". The URL pattern is "{controller}/{action}/{id}", and the defaults are set to controller = "Home", action = "Index", and id = UrlParameter.Optional. The file is named **RouteConfig.cs**.



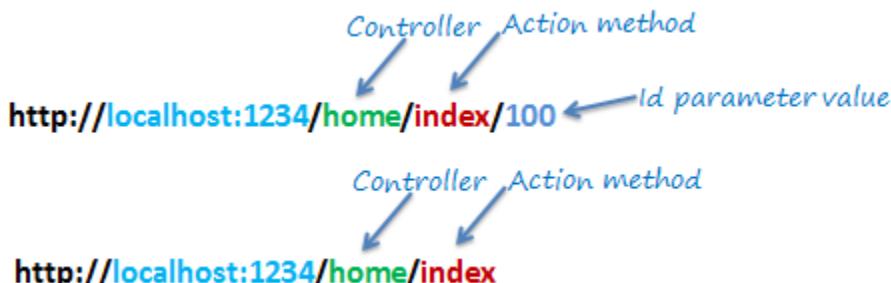
Configure Route in MVC

As you can see in the above figure, the route is configured using the MapRoute() extension method of RouteCollection, where name is "Default", url pattern is "`{controller}/{action}/{id}`" and defaults parameter for controller, action method and id parameter. Defaults specifies which controller, action method or value of id parameter should be used if they do not exist in the incoming request URL.

The same way, you can configure other routes using MapRoute method of RouteCollection. This RouteCollection is actually a property of [RouteTable](#) class.

URL Pattern

The URL pattern is considered only after domain name part in the URL. For example, the URL pattern "`{controller}/{action}/{id}`" would look like localhost:1234/{controller}/{action}/{id}. Anything after "localhost:1234/" would be considered as controller name. The same way, anything after controller name would be considered as action name and then value of id parameter.



Routing in MVC

If the URL doesn't contain anything after domain name then the default controller and action method will handle the request. For example, `http://localhost:1234` would be handled by HomeController and Index method as configured in the defaults parameter.

The following table shows which Controller, Action method and Id parameter would handle different URLs considering above default route.

URL	Controller	Action	Id
<code>http://localhost/home</code>	HomeController	Index	null
<code>http://localhost/home/index/123</code>	HomeController	Index	123
<code>http://localhost/home/about</code>	HomeController	About	null
<code>http://localhost/home/contact</code>	HomeController	Contact	null

URL	Controller	Action	Id
http://localhost/student	StudentController	Index	null
http://localhost/student/edit/123	StudentController	Edit	123

Multiple Routes

You can also configure a custom route using MapRoute extension method. You need to provide at least two parameters in MapRoute, route name and url pattern. The Defaults parameter is optional.

You can register multiple custom routes with different names. Consider the following example where we register "Student" route.

Example: Custom Routes

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Student",
            url: "students/{id}",
            defaults: new { controller = "Student", action = "Index" }
        );

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
        );
    }
}
```

As shown in the above code, URL pattern for the Student route is **students/{id}**, which specifies that any URL that starts with domainName/students, must be handled by StudentController. Notice that we haven't specified {action} in the URL pattern because we want every URL that starts with student should always use Index action of StudentController. We have specified default controller and action to handle any URL request which starts from domainname/students.

MVC framework evaluates each route in sequence. It starts with first configured route and if incoming url doesn't satisfy the URL pattern of the route then it will evaluate second route and so on. In the above example, routing engine will evaluate Student route first and if incoming url doesn't

starts with /students then only it will consider second route which is default route.

The following table shows how different URLs will be mapped to Student route:

URL	Controller	Action	Id
http://localhost/student/123	StudentController	Index	123
http://localhost/student/index/123	StudentController	Index	123
http://localhost/student?Id=123	StudentController	Index	123

Route Constraints

You can also apply restrictions on the value of parameter by configuring route constraints. For example, the following route applies a restriction on id parameter that the value of an id must be numeric.

Example: Route Constraints

```
routes.MapRoute(
    name: "Student",
    url: "student/{id}/{name}/{standardId}",
    defaults: new { controller = "Student", action = "Index", id =
UrlParameter.Optional, name = UrlParameter.Optional, standardId =
UrlParameter.Optional },
    constraints: new { id = @"\d+" })
);
```

So if you give non-numeric value for id parameter then that request will be handled by another route or, if there are no matching routes then "*The resource could not be found*" error will be thrown.

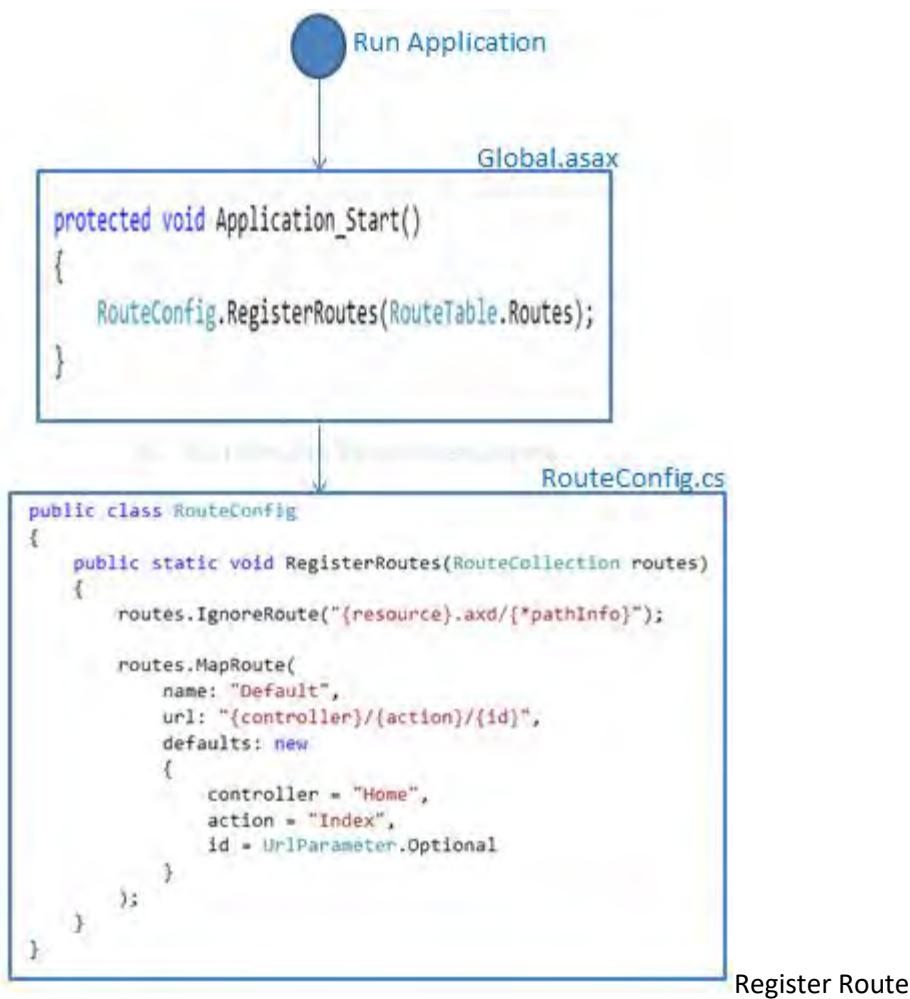
Register Routes

Now, after configuring all the routes in RouteConfig class, you need to register it in the Application_Start() event in the Global.asax. So that it includes all your routes into RouteTable.

Example: Route Registration

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        RouteConfig.RegisterRoutes(RouteTable.Routes);
    }
}
```

The following figure illustrate Route registration process.



Thus, routing plays important role in MVC framework.

Controller

In this section, you will learn about the Controller in ASP.NET MVC.

The Controller in MVC architecture handles any incoming URL request. Controller is a class, derived from the base class `System.Web.Mvc.Controller`. Controller class contains public methods called **Action** methods. Controller and its action method handles incoming browser requests, retrieves necessary model data and returns appropriate responses.

In ASP.NET MVC, every controller class name must end with a word "Controller". For example, controller for home page must be `HomeController` and controller for student must be `StudentController`. Also, every controller class must be located in Controller folder of MVC folder structure.

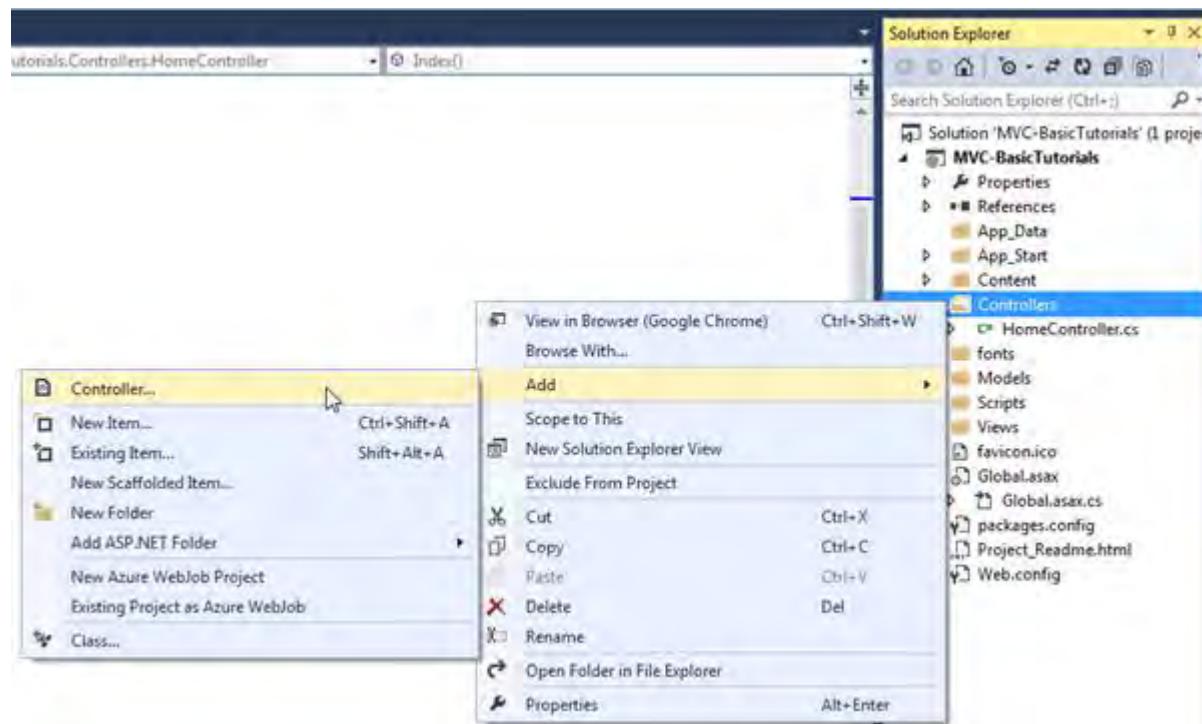
Adding a New Controller

Now, let's add a new empty controller in our MVC application in Visual Studio.

MVC will throw "The resource cannot be found" error when you do not append "Controller" to the controller class name.

In the previous section we learned how to create our first MVC application, which in turn created a default HomeController. Here, we will create a new StudentController.

In the Visual Studio, right click on the Controller folder -> select **Add** -> click on **Controller..**

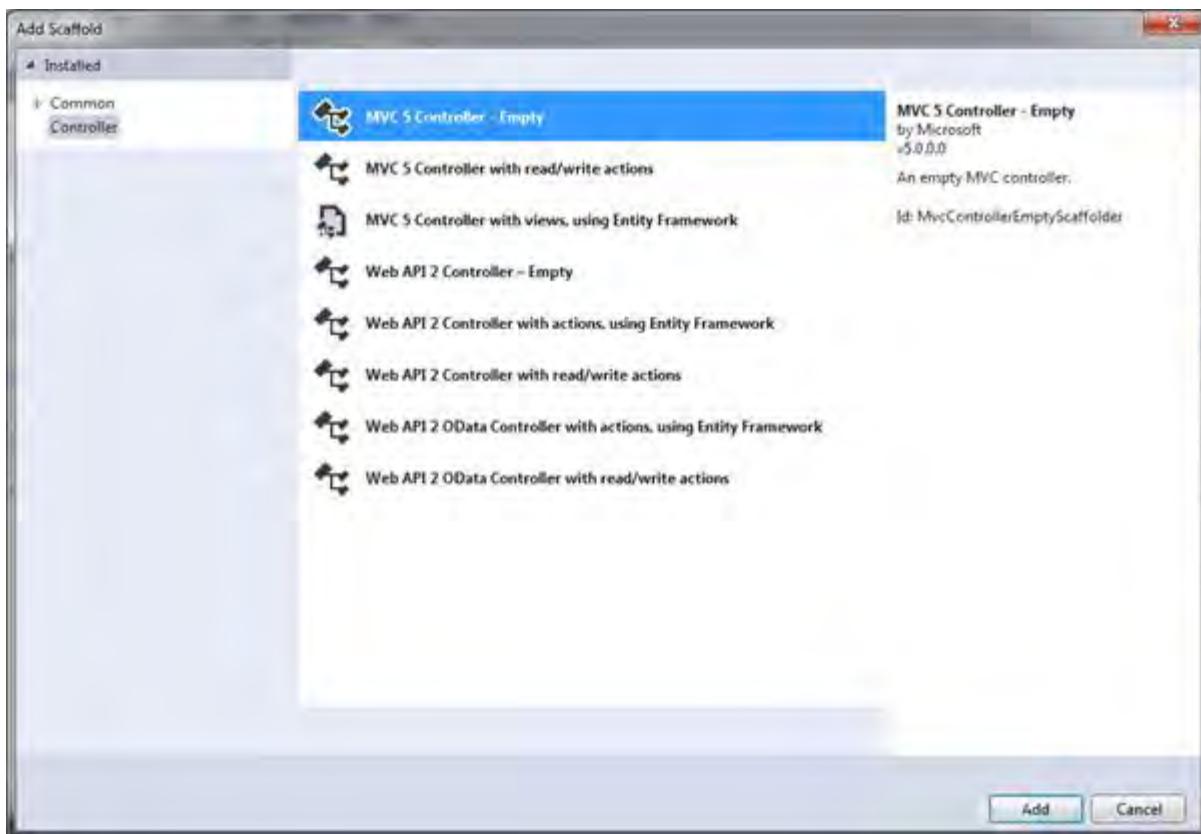


Add New Controller

This opens Add Scaffold dialog as shown below.

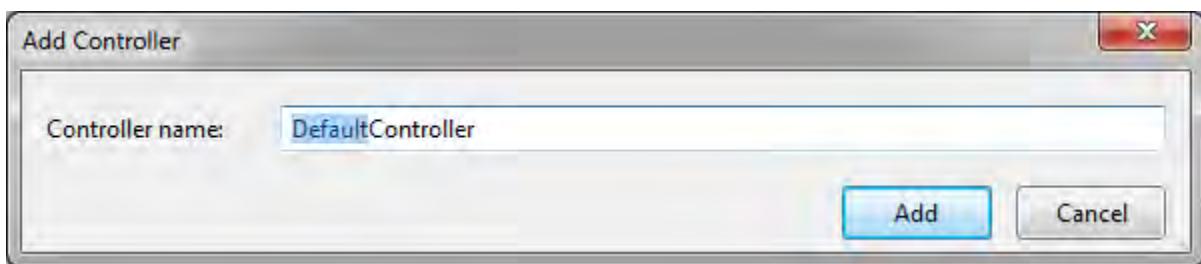
Note:

Scaffolding is an automatic code generation framework for ASP.NET web applications. Scaffolding reduces the time taken to develop a controller, view etc. in MVC framework. You can develop a customized scaffolding template using T4 templates as per your architecture and coding standard.



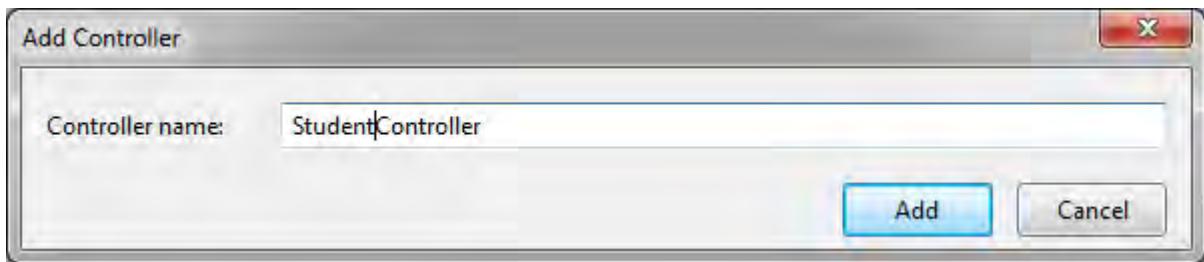
Adding Controller

Add Scaffold dialog contains different templates to create a new **controller**. We will learn about other templates later. For now, select "**MVC 5 Controller - Empty**" and click **Add**. It will open Add Controller dialog as shown below



Adding Controller

In the Add Controller dialog, enter the name of the controller. Remember, controller name must end with Controller. Let's enter StudentController and click **Add**.



Adding Controller

This will create StudentController class with Index method in StudentController.cs file under Controllers folder, as shown below.

Example: Controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

As you can see above, the StudentController class is derived from Controller class. Every controller in MVC must derived from this abstract Controller class. This base Controller class contains helper methods that can be used for various purposes.

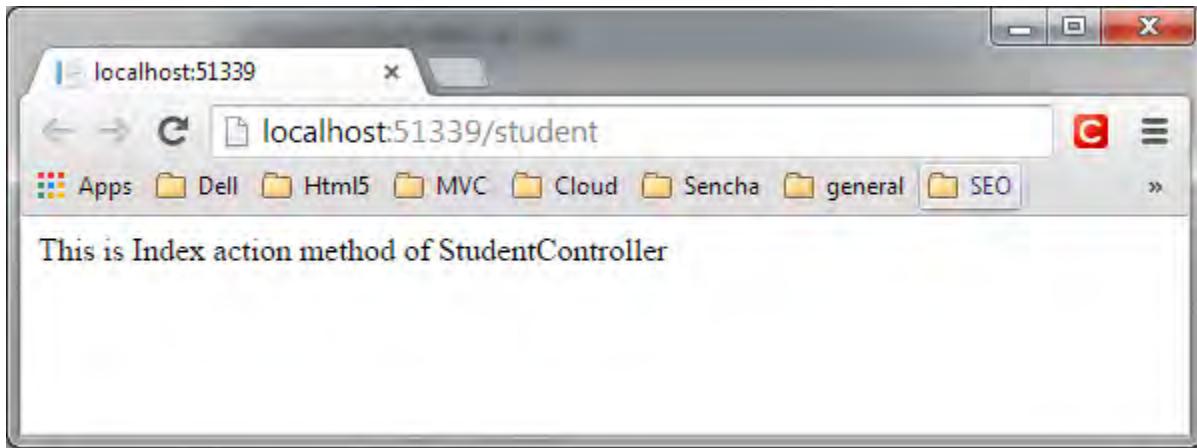
Now, we will return a dummy string from Index action method of above StudentController. Changing the return type of Index method from ActionResult to string and returning dummy string is shown below. You will learn about ActionResult in the next section.

Example: Controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
```

```
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public string Index()
        {
            return "This is Index action method of StudentController";
        }
    }
}
```

We have already seen in the routing section that the URL request `http://localhost/student` or `http://localhost/student/index` is handled by the `Index()` method of `StudentController` class, shown above. So let's invoke it from the browser and you will see the following page in the browser.



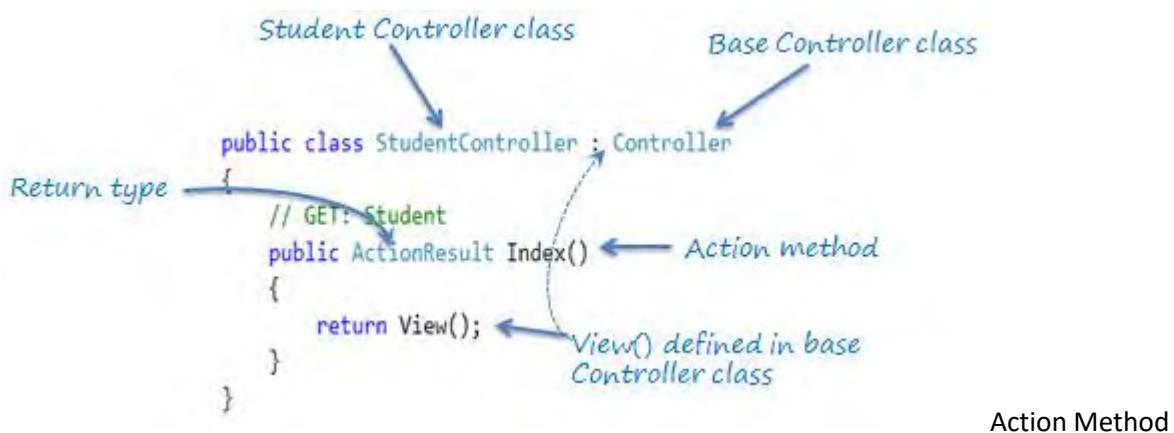
Action method

In this section, you will learn about the action method of controller class.

All the public methods of a Controller class are called Action methods. They are like any other normal methods with the following restrictions:

1. Action method must be public. It cannot be private or protected
2. Action method cannot be overloaded
3. Action method cannot be a static method.

The following is an example of Index action method of `StudentController`



As you can see in the above figure, Index method is a public method and it returns ActionResult using the View() method. The View() method is defined in the Controller base class, which returns the appropriate ActionResult.

Default Action Method

Every controller can have default action method as per configured route in RouteConfig class. By default, Index is a default action method for any controller, as per configured default root as shown below.

Default Route:

```

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{name}",
    defaults: new { controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
    });

```

However, you can change the default action name as per your requirement in RouteConfig class.

ActionResult

MVC framework includes various result classes, which can be return from an action methods. There result classes represent different types of responses such as html, file, string, json, javascript etc. The following table lists all the result classes available in ASP.NET MVC.

Result Class	Description
ViewResult	Represents HTML and markup.
EmptyResult	Represents No response.

Result Class	Description
ContentResult	Represents string literal.
FileContentResult/ FilePathResult/ FileStreamResult	Represents the content of a file
JavaScriptResult	Represent a JavaScript script.
JsonResult	Represent JSON that can be used in AJAX
RedirectResult	Represents a redirection to a new URL
RedirectToRouteResult	Represent another action of same or other controller
PartialViewResult	Returns HTML from Partial view
HttpUnauthorizedResult	Returns HTTP 403 status

The ActionResult class is a base class of all the above result classes, so it can be return type of action methods which returns any type of result listed above. However, you can specify appropriate result class as a return type of action method.

The Index() method of StudentController in the above figure uses View() method to return ViewResult (which is derived from ActionResult). The View() method is defined in base Controller class. It also contains different methods, which automatically returns particular type of result as shown in the below table.

Result Class	Description	Base Controller Method
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file	File()
JavaScriptResult	Represent a JavaScript script.	JavaScript()
JsonResult	Represent JSON that can be used in AJAX	Json()
RedirectResult	Represents a redirection to a new URL	Redirect()

Result Class	Description	Base Controller Method
RedirectToRouteResult	Represent another action of same or other controller	RedirectToRoute()
PartialViewResult	Returns HTML	PartialView()
HttpUnauthorizedResult	Returns HTTP 403 status	

As you can see in the above table, View method returns ViewResult, Content method returns string, File method returns content of a file and so on. Use different methods mentioned in the above table, to return different types of results from an action method.

Action Method Parameters

Every action methods can have input parameters as normal methods. It can be primitive data type or complex type parameters as shown in the below example.

Example: Action method parameters

```
[HttpPost]
public ActionResult Edit(Student std)
{
    // update student to the database

    return RedirectToAction("Index");
}

[HttpDelete]
public ActionResult Delete(int id)
{
    // delete student from the database whose id matches with specified id

    return RedirectToAction("Index");
}
```

Please note that action method parameter can be [Nullable Type](#).

By default, the values for action method parameters are retrieved from the request's data collection. The data collection includes name/values pairs for form data or query string values or cookie values. Model binding in ASP.NET MVC automatically maps the URL query string or form data collection to the action method parameters if both names are matching.

Action Selectors

Action selector is the attribute that can be applied to the action methods. It helps routing engine to select the correct action method to handle a particular request. MVC 5 includes the following action selector attributes:

1. ActionName
2. NonAction
3. ActionVerbs

ActionName

ActionName attribute allows us to specify a different action name than the method name. Consider the following example.

Example: ActionName

```
public class StudentController : Controller
{
    public StudentController()
    {

    }

    [ActionName("find")]
    public ActionResult GetById(int id)
    {
        // get student from the database
        return View();
    }
}
```

In the above example, we have applied `ActionName("find")` attribute to `GetById` action method. So now, action name is "find" instead of "GetById". This action method will be invoked on `http://localhost/student/find/1` request instead of `http://localhost/student/getbyid/1` request.

NonAction

NonAction selector attribute indicates that a public method of a Controller is not an action method. Use NonAction attribute when you want public method in a controller but do not want to treat it as an action method.

For example, the `GetStudent()` public method cannot be invoked in the same way as action method in the following example.

Example: NonAction

```
public class StudentController : Controller
{
    public StudentController()
    {

    }

    [NonAction]
    public Student GetStudent(int id)
    {
        return studentList.Where(s => s.StudentId == id).FirstOrDefault();
    }
}
```

ActionVerbs

In this section, you will learn about the ActionVerbs selectors attribute.

The ActionVerbs selector is used when you want to control the selection of an action method based on a Http request method. For example, you can define two different action methods with the same name but one action method responds to an HTTP Get request and another action method responds to an HTTP Post request.

MVC framework supports different ActionVerbs, such as `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete`, `HttpOptions` & `HttpPatch`. You can apply these attributes to action method to indicate the kind of Http request the action method supports. If you do not apply any attribute then it considers it a GET request by default.

The following figure illustrates the `HttpGET` and `HttpPOST` action verbs.

```

http://localhost/Student/Edit/1
    HttpGET →
        public ActionResult Edit(int Id)
        {
            var std = students.Where(s => s.StudentId == Id).FirstOrDefault();

            return View(std);
        }

http://localhost/Student/Edit [HttpPost]
    HttpPOST →
        public ActionResult Edit(Student std)
        {
            //update database here..

            return RedirectToAction("Index");
        }

```

ActionVerbs

The following table lists the usage of http methods:

Http method	Usage
GET	To retrieve the information from the server. Parameters will be appended in the query string.
POST	To create a new resource.
PUT	To update an existing resource.
HEAD	Identical to GET except that server do not return message body.
OPTIONS	OPTIONS method represents a request for information about the communication options supported by web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

Visit [W3.org](https://www.w3.org/) for more information on Http Methods.

The following example shows different action methods supports different ActionVerbs:

Example: ActionVerbs

```

public class StudentController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}

```

```

}

[HttpPost]
public ActionResult PostAction()
{
    return View("Index");
}

[HttpPut]
public ActionResult PutAction()
{
    return View("Index");
}

[HttpDelete]
public ActionResult DeleteAction()
{
    return View("Index");
}

[HttpHead]
public ActionResult HeadAction()
{
    return View("Index");
}

[HttpOptions]
public ActionResult OptionsAction()
{
    return View("Index");
}

[HttpPatch]
public ActionResult PatchAction()
{
    return View("Index");
}
}

```

You can also apply multiple http verbs using `AcceptVerbs` attribute. `GetAndPostAction` method supports both, GET and POST ActionVerbs in the following example:

Example: AcceptVerbs

```

[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Get)]
public ActionResult GetAndPostAction()
{
    return RedirectToAction("Index");
}

```

Model in ASP.NET MVC

In this section, you will learn about the Model in ASP.NET MVC framework.

Model represents domain specific data and business logic in MVC architecture. It maintains the data of the application. Model objects retrieve and store model state in the persistence like a database.

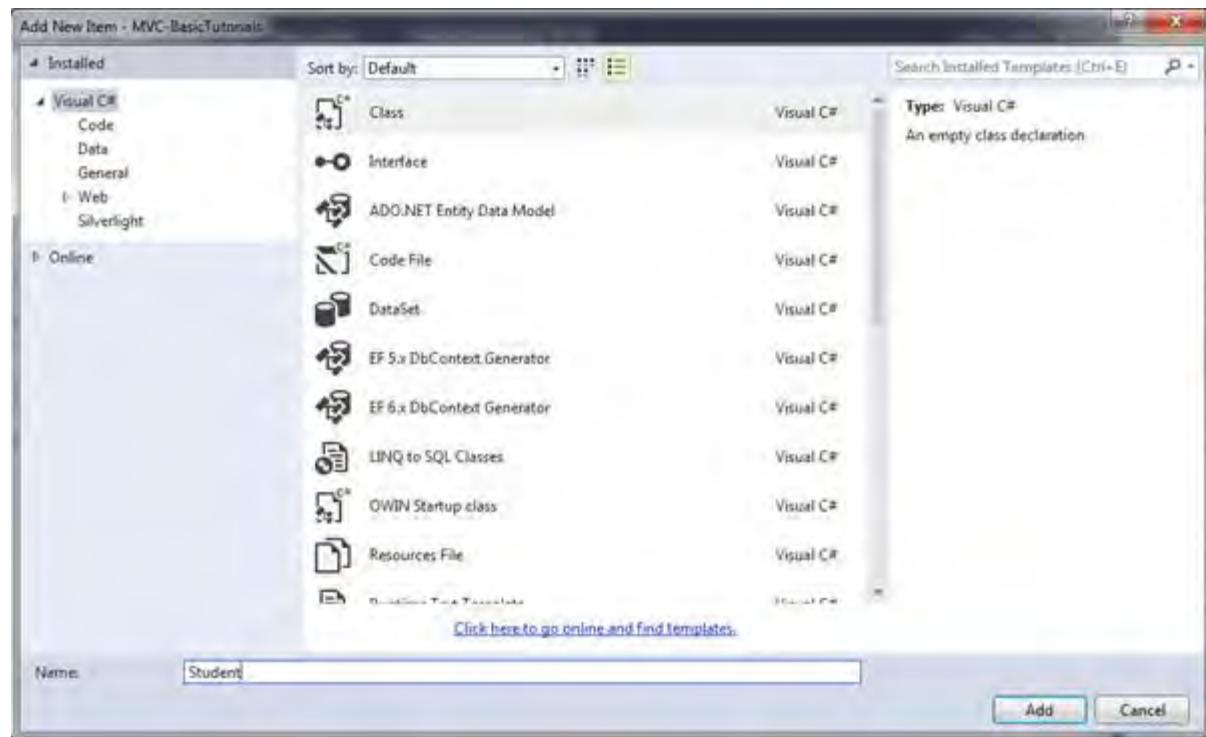
Model class holds data in public properties. All the Model classes reside in the Model folder in MVC folder structure.

Let's see how to add model class in ASP.NET MVC.

Adding a Model

Open our first MVC project created in previous step in the Visual Studio. Right click on Model folder -> Add -> click on Class..

In the Add New Item dialog box, enter class name 'Student' and click **Add**.



Create Model Class

This will add new Student class in model folder. Now, add Id, Name, Age properties as shown below.

Example: Model class

```
namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set; }
        public int Age { get; set; }
    }
}
```

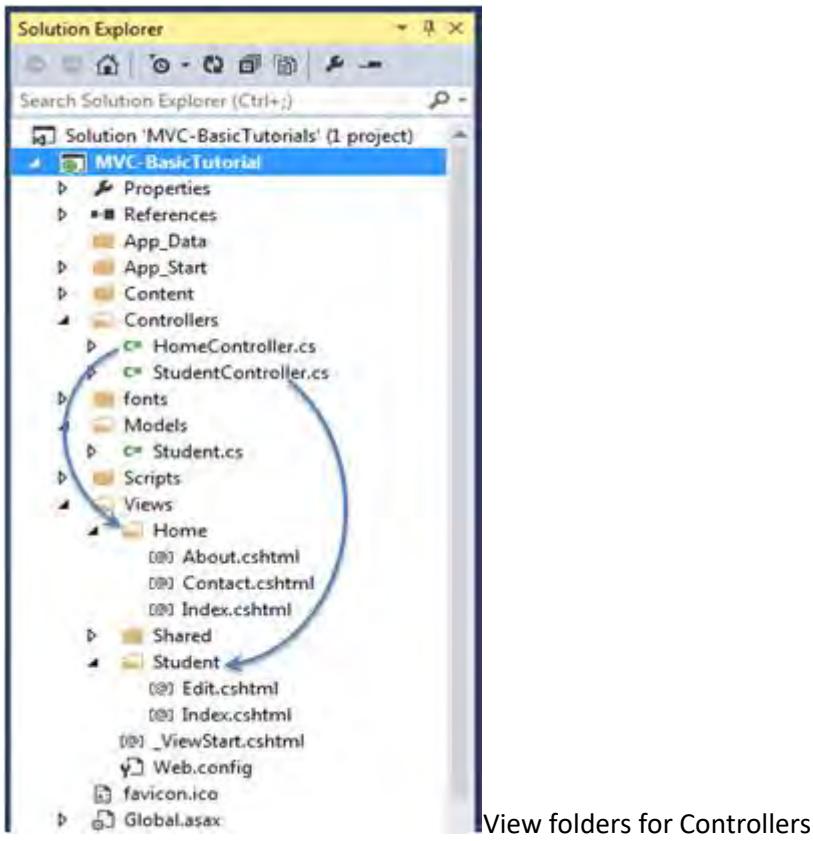
So in this way, you can create a model class which you can use in View. You will learn how to implement validations using model later.

View in ASP.NET MVC

In this section, you will learn about the View in ASP.NET MVC framework.

View is a user interface. View displays data from the model to the user and also enables them to modify the data.

ASP.NET MVC views are stored in Views folder. Different action methods of a single controller class can render different views, so the Views folder contains a separate folder for each controller with the same name as controller, in order to accommodate multiple views. For example, views, which will be rendered from any of the action methods of HomeController, resides in Views > Home folder. In the same way, views which will be rendered from StudentController, will resides in Views > Student folder as shown below.



View folders for Controllers

Note:

Shared folder contains views, layouts or partial views which will be shared among multiple views.

Razor View Engine

Microsoft introduced the Razor view engine and packaged with MVC 3. You can write a mix of html tags and server side code in razor view. Razor uses @ character for server side code instead of traditional <% %>. You can use C# or Visual Basic syntax to write server side code inside razor view. Razor view engine maximize the speed of writing code by minimizing the number of characters and keystrokes required when writing a view. Razor views files have .cshtml or vbhtml extension.

ASP.NET MVC supports following types of view files:

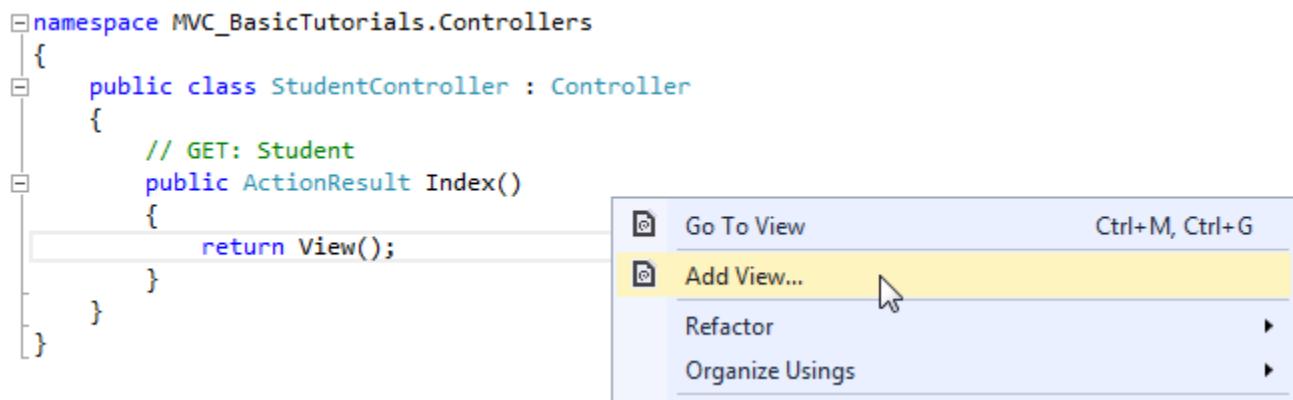
View file extension	Description
.cshtml	C# Razor view. Supports C# with html tags.
.vbhtml	Visual Basic Razor view. Supports Visual Basic with html tags.

View file extension	Description
.aspx	ASP.Net web form
.ascx	ASP.NET web control

Create New View

We have already created StudentController and Student model in the previous section. Now, let's create a Student view and understand how to use model into view.

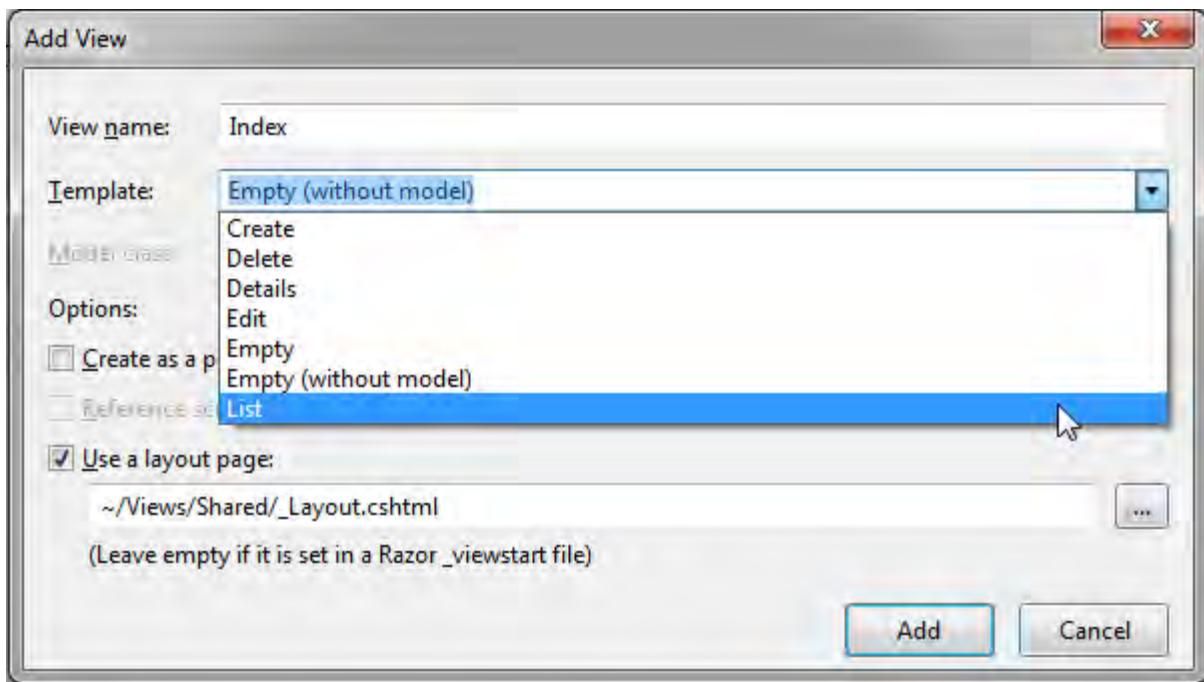
We will create a view, which will be rendered from Index method of StudentController. So, open a StudentController class -> right click inside Index method -> click **Add View..**



Create a View

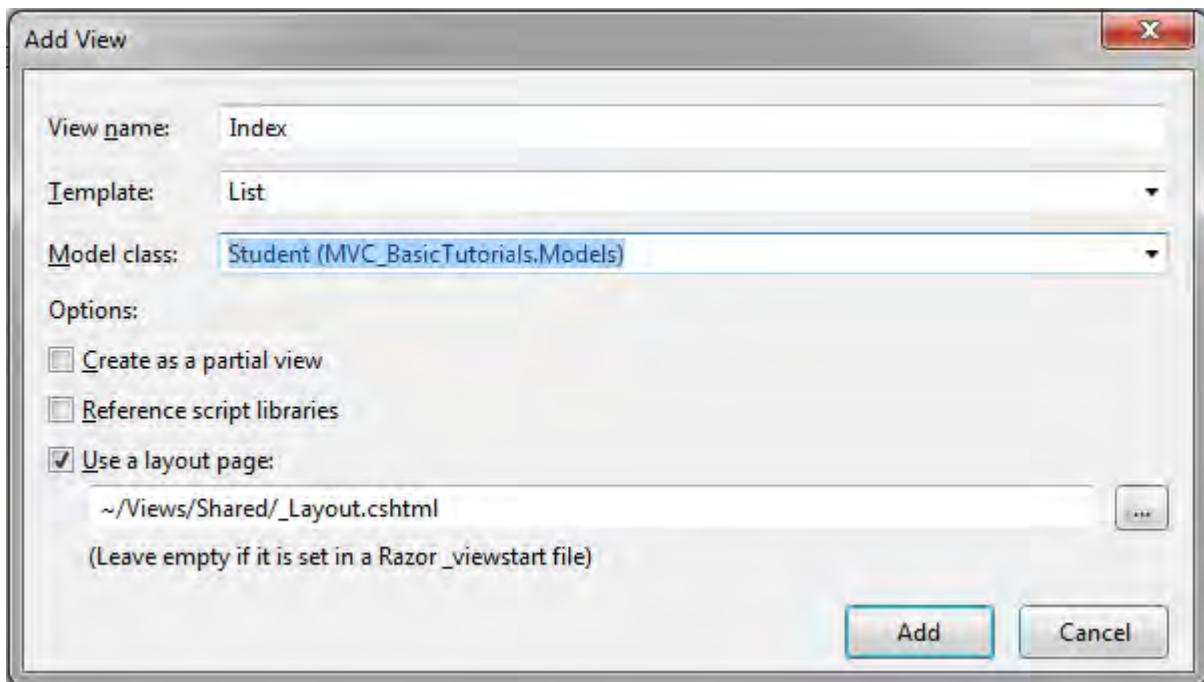
In the Add View dialogue box, keep the view name as Index. It's good practice to keep the view name the same as the action method name so that you don't have to specify view name explicitly in the action method while returning the view.

Select the scaffolding template. Template dropdown will show default templates available for Create, Delete, Details, Edit, List or Empty view. Select "List" template because we want to show list of students in the view.



View

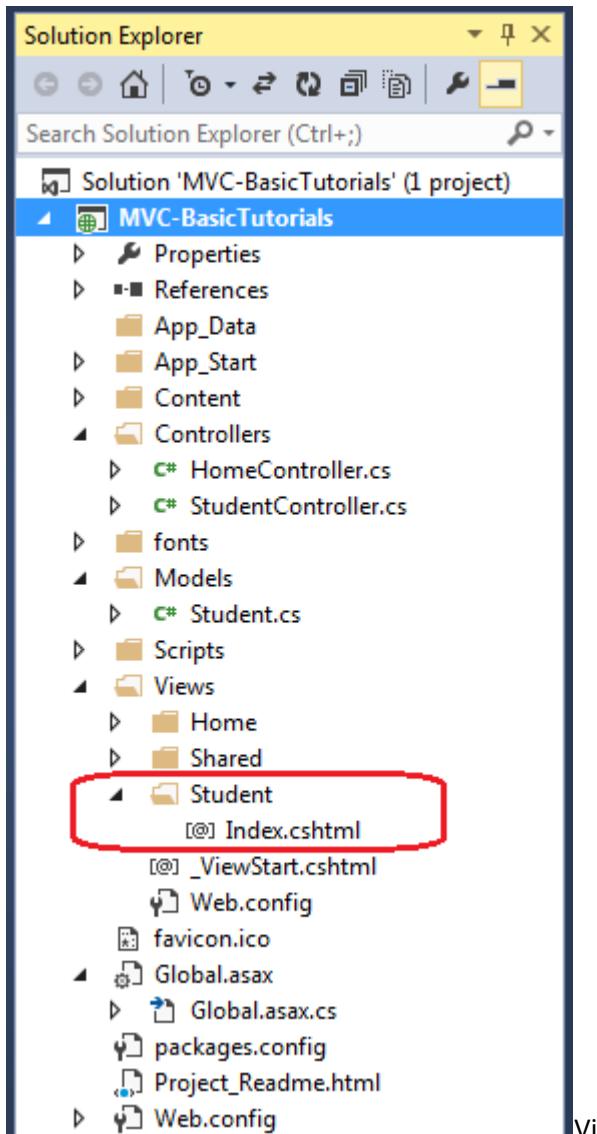
Now, select Student from the Model class dropdown. Model class dropdown automatically displays the name of all the classes in the Model folder. We have already created Student Model class in the previous section, so it would be included in the dropdown.



View

Check "Use a layout page" checkbox and select _Layout.cshtml page for this view and then click **Add** button. We will see later what is layout page but for now think it like a master page in MVC.

This will create Index view under View -> Student folder as shown below:



The following code snippet shows an Index.cshtml created above.

Views\Student\Index.cshtml:

```
@model IEnumerable<MVC_BasicTutorials.Models.Student>
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
```

```

}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Age)
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.StudentName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Age)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |
                @Html.ActionLink("Details", "Details", new { id=item.StudentId }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.StudentId })
            </td>
        </tr>
    }
</table>

```

As you can see in the above Index view, it contains both Html and razor codes. Inline razor expression starts with @ symbol. @Html is a helper class to generate html controls. You will learn razor syntax and html helpers in the coming sections.

```

@model IEnumerable<MVC_BasicTutorials.Models.Student>
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Index</h2>
<p>@Html.ActionLink("Create New", "Create")</p>
<table class="table">
    <thead>
        <tr>
            <th>@Html.DisplayNameFor(model => model.StudentName)</th>
            <th>@Html.DisplayNameFor(model => model.Age)</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>@Html.DisplayFor(modelItem => item.StudentName)</td>
                <td>@Html.DisplayFor(modelItem => item.Age)</td>
                <td>@Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |</td>
            </tr>
        }
    </tbody>
</table>

```

Razor Syntax

Html

Html helper

Index.cshtml

The above Index view would look like below.

Index

[Create New](#)

Name	Age	
John	18	Edit Details Delete
Steve	21	Edit Details Delete
Bill	25	Edit Details Delete
Ram	20	Edit Details Delete
Ron	31	Edit Details Delete
Chris	17	Edit Details Delete
Rob	19	Edit Details Delete

© 2014 - My ASP.NET Application

Index View

Note:

Every view in the ASP.NET MVC is derived from WebViewPage class included in System.Web.Mvc namespace.

Integrate Controller, View and Model

We have already created StudentController, model and view in the previous sections, but we have not integrated all these components in-order to run it.

The following code snippet shows StudentController and Student model class & view created in the previous sections.

Example: StudentController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```

using System.Web.Mvc;
using MVC_BasicTutorials.Models;

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}

```

Example: Student Model class

```

namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set; }
        public int Age { get; set; }
    }
}

```

Example: Index.cshtml to display student list

```

@model IEnumerable<MVC_BasicTutorials.Models.Student>

 @{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}



## Index



Create New



| @Html.DisplayNameFor(model => model.StudentName) | @Html.DisplayNameFor(model => model.Age) |
|--------------------------------------------------|------------------------------------------|
|--------------------------------------------------|------------------------------------------|


```

```

        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Age)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |
            @Html.ActionLink("Details", "Details", new { id=item.StudentId }) |
            @Html.ActionLink("Delete", "Delete", new { id = item.StudentId })
        </td>
    </tr>
}
</table>

```

Now, to run it successfully, we need to pass a model object from controller to Index view. As you can see in the above Index.cshtml, it uses IEnumerable of Student as a model object. So we need to pass IEnumerable of Student model from the Index action method of StudentController class as shown below.

Example: Passing Model from Controller

```

public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        var studentList = new List<Student>{
            new Student() { StudentId = 1, StudentName = "John", Age
= 18 } ,
            new Student() { StudentId = 2, StudentName = "Steve", Age
= 21 } ,
            new Student() { StudentId = 3, StudentName = "Bill", Age
= 25 } ,
            new Student() { StudentId = 4, StudentName = "Ram" , Age
= 20 } ,
            new Student() { StudentId = 5, StudentName = "Ron" , Age
= 31 } ,
            new Student() { StudentId = 4, StudentName = "Chris" , Age
= 17 } ,
            new Student() { StudentId = 4, StudentName = "Rob" , Age
= 19 }
        };
        // Get the students from the database in the real application

        return View(studentList);
    }
}

```

As you can see in the above code, we have created a List of student objects for an example purpose (in real life application, you can fetch it from the database). We then pass this list object as a parameter in the View() method. The View() method is defined in base Controller class, which automatically binds model object to the view.

Now, you can run the MVC project by pressing F5 and navigate to <http://localhost/Student>. You will see following view in the browser.

The screenshot shows a web page titled "Application name" with a navigation bar including "Home", "About", and "Contact". Below the navigation bar is a section titled "Index" with a "Create New" link. The main content is a table listing student names and ages, each with edit, details, and delete links. The table has columns for "Name" and "Age". The data is as follows:

Name	Age	
John	18	Edit Details Delete
Steve	21	Edit Details Delete
Bill	25	Edit Details Delete
Ram	20	Edit Details Delete
Ron	31	Edit Details Delete
Chris	17	Edit Details Delete
Rob	19	Edit Details Delete

At the bottom of the page, there is a copyright notice: "© 2014 - My ASP.NET Application".

Model Binding

In this section, you will learn about model binding in MVC framework.

To understand the model binding in MVC, first let's see how you can get the http request values in the action method using traditional ASP.NET style. The following figure shows how you can get the values from HttpGET and HttpPOST request by using the Request object directly in the action method.

```

    HttpGET →
/Student/Edit?id=1

public ActionResult Edit()
{
    var id = Request.QueryString["id"];

    // retrieve data from the database

    return View();
}

[HttpPost]
HttpPOST →
/Student/Edit

public ActionResult Edit()
{
    var id = Request["StudentId"];
    var name = Request["StudentName"];
    var age = Request["Age"];

    //update database here..

    return RedirectToAction("Index");
}

```

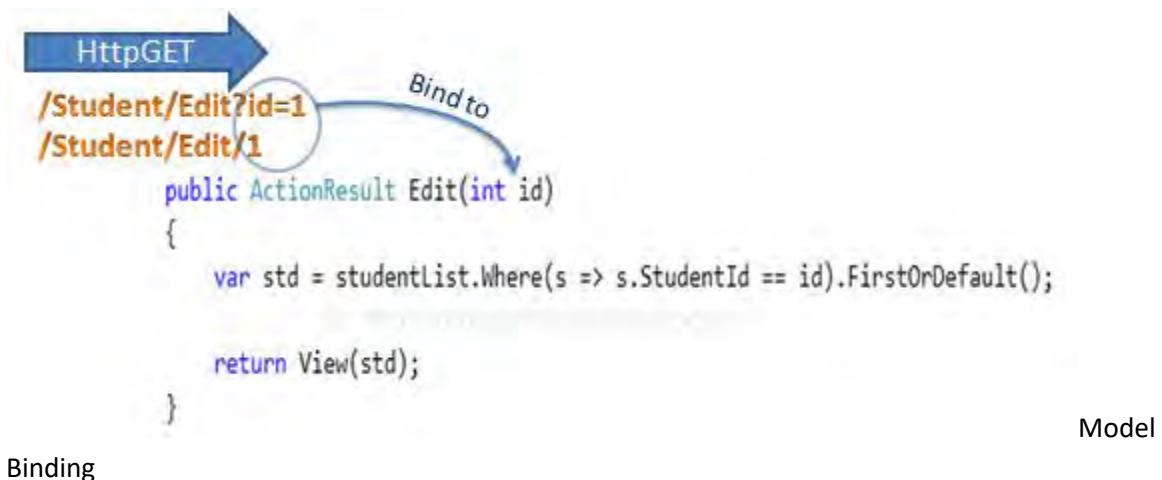
Accessing Request Data

As you can see in the above figure, we use the `Request.QueryString` and `Request(Request.Form)` object to get the value from `HttpGet` and `HttpPost` request. Accessing request values using the `Request` object is a cumbersome and time wasting activity.

With model binding, MVC framework converts the http request values (from query string or form collection) to action method parameters. These parameters can be of primitive type or complex type.

Binding to Primitive type

`HttpGet` request embeds data into a query string. MVC framework automatically converts a query string to the action method parameters. For example, the query string "id" in the following GET request would automatically be mapped to the `id` parameter of the `Edit()` action method.



TIPS

This binding is case insensitive. So "id" parameter can be "ID" or "Id".

You can also have multiple parameters in the action method with different data types. Query string values will be converted into parameters based on matching name.

For example, `http://localhost/Student/Edit?id=1&name=John` would map to id and name parameter of the following Edit action method.

Example: Convert QueryString to Action Method Parameters

```

public ActionResult Edit(int id, string name)
{
    // do something here
    return View();
}

```

Binding to Complex type

Model binding also works on complex types. Model binding in MVC framework automatically converts form field data of HttpPOST request to the properties of a complex type parameter of an action method.

Consider the following model classes.

Example: Model classes in C#

```

public class Student
{
    public int StudentId { get; set; }
}

```

```

[Display(Name="Name")]
public string StudentName { get; set; }
public int Age { get; set; }
public Standard standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }
}

```

Now, you can create an action method which includes Student type parameter. In the following example, Edit action method (HttpPost) includes Student type parameter.

Example: Action Method with Class Type Parameter

```

[HttpPost]
public ActionResult Edit(Student std)
{
    var id = std.StudentId;
    var name = std.StudentName;
    var age = std.Age;
    var standardName = std.standard.StandardName;

    //update database here..

    return RedirectToAction("Index");
}

```

So now, MVC framework will automatically maps Form collection values to Student type parameter when the form submits http POST request to Edit action method as shown below.

```

[HttpPost]
public ActionResult Edit(Student std)
{
    var id = std.StudentId;
    var name = std.StudentName;
    var age = std.Age;
    var standardName = std.standard.StandardName;

    //update database here..
    return RedirectToAction("Index");
}

```

Model Binding

So thus, it automatically binds form fields to the complex type parameter of action method.

FormCollection

You can also include FormCollection type parameter in the action method instead of complex type, to retrieve all the values from view form fields as shown below.



Model Binding

Bind Attribute

ASP.NET MVC framework also enables you to specify which properties of a model class you want to bind. The [Bind] attribute will let you specify the exact properties a model binder should include or exclude in binding.

In the following example, Edit action method will only bind StudentId and StudentName property of a Student model.

Example: Binding Parameters

```
[HttpPost]
public ActionResult Edit([Bind(Include = "StudentId, StudentName")] Student std)
{
    var name = std.StudentName;

    //write code to update student

    return RedirectToAction("Index");
}
```

You can also use Exclude properties as below.

Example: Exclude Properties in Binding

```
[HttpPost]
public ActionResult Edit([Bind(Exclude = "Age")] Student std)
{
    var name = std.StudentName;

    //write code to update student

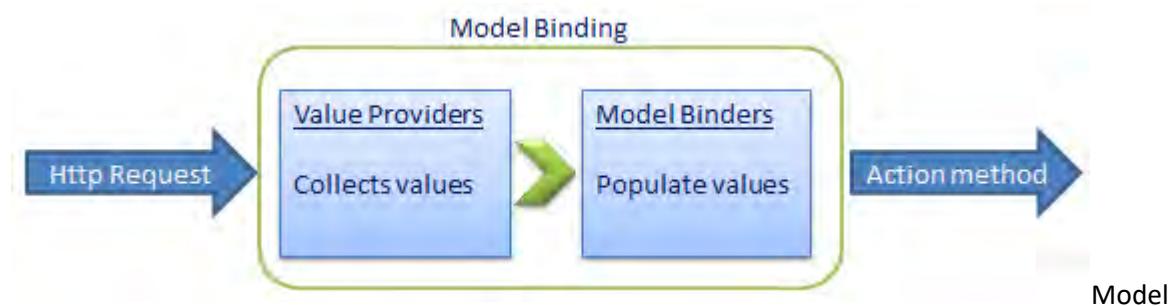
    return RedirectToAction("Index");
}
```

The Bind attribute will improve the performance by only bind properties which you needed.

Inside Model Binding

As you have seen that Model binding automatically converts request values into a primitive or complex type object. Model binding is a two step process. First, it collects values from the incoming http request and second, populates primitive type or complex type with these values.

Value providers are responsible for collecting values from request and Model Binders are responsible for populating values.



Binding in MVC

Default value provider collection evaluates values from the following sources:

1. Previously bound action parameters, when the action is a child action
2. Form fields (Request.Form)
3. The property values in the JSON Request body (Request.InputStream), but only when the request is an AJAX request
4. Route data (RouteData.Values)
5. Querystring parameters (Request.QueryString)
6. Posted files (Request.Files)

MVC includes [DefaultModelBinder](#) class which effectively binds most of the model types.

Visit MSDN for detailed information on [Model binding](#)

Returning/Rendering a View

View()

Generate HTML to display for the required view and render/send it to the browser.

```
public ActionResult Index()
{
    UserDetailModel obj = new UserDetailModel();
    obj.CountryDetail = new List<Country>()
    {
        .....
    };
    return View(obj);
}
```

Redirect()

Redirect to specified URL instead of rendering HTML
(ie., it redirect to new URL).

```
[HttpPost]  
public ActionResult Index(string Name)  
{  
    return Redirect("Home/MyIndex");  
}
```

RedirectToAction()

Redirect to specified action instead of rendering HTML.

```
[HttpPost]  
public ActionResult Index(string Name)  
{  
    return RedirectToAction("MyIndex");  
}
```

RedirectToRoute()

- Look up the specified route into the Route table that is defined in global.asax and then redirect to that controller/action defined in that route. This also make a new request like RedirectToAction().

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapRoute(
        "MyRoute", // Route name
        "Account/", // URL
        new { controller = "Account", action = "Login" } // Parameter defaults
    );
}
```

RedirectToRoute()

```
routes.MapRoute(  
    "Default", // Route name  
    "{controller}/{action}/{id}", // URL with parameters  
    new { controller = "Home", action = "MyIndex", id = UrlParameter.Optional } //  
    Parameter defaults  
);  
}
```

RedirectToRoute()

HomeController

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

[HttpPost]
public ActionResult Index(string Name)
{
    return RedirectToRoute("MyRoute");
}
```

RedirectToRoute()

AccountController

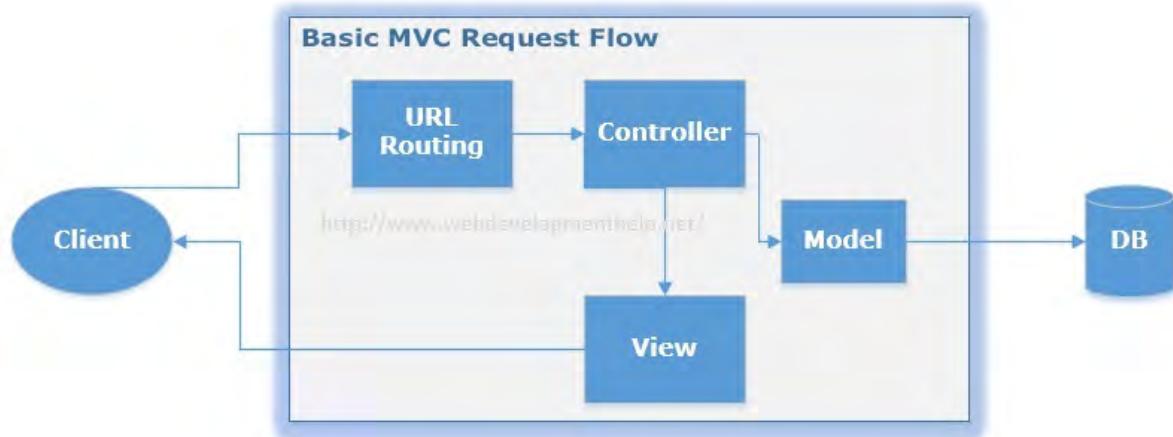
```
public ActionResult Login()
{
    return View();
}
```

Summary

- Return View doesn't make a new requests, it just renders the view without changing URLs in the browser's address bar.
- Return RedirectToAction makes a new requests and URL in the browser's address bar is updated with the generated URL by MVC.
- Return Redirect also makes a new requests and URL in the browser's address bar is updated, but you have to specify the full URL to redirect
- Between RedirectToAction and Redirect, best practice is to use RedirectToAction for anything dealing with your application actions/controllers. If you use Redirect and provide the URL, you'll need to modify those URLs manually when you change the route table.
- RedirectToRoute redirects to a specific route defined in the Route table.

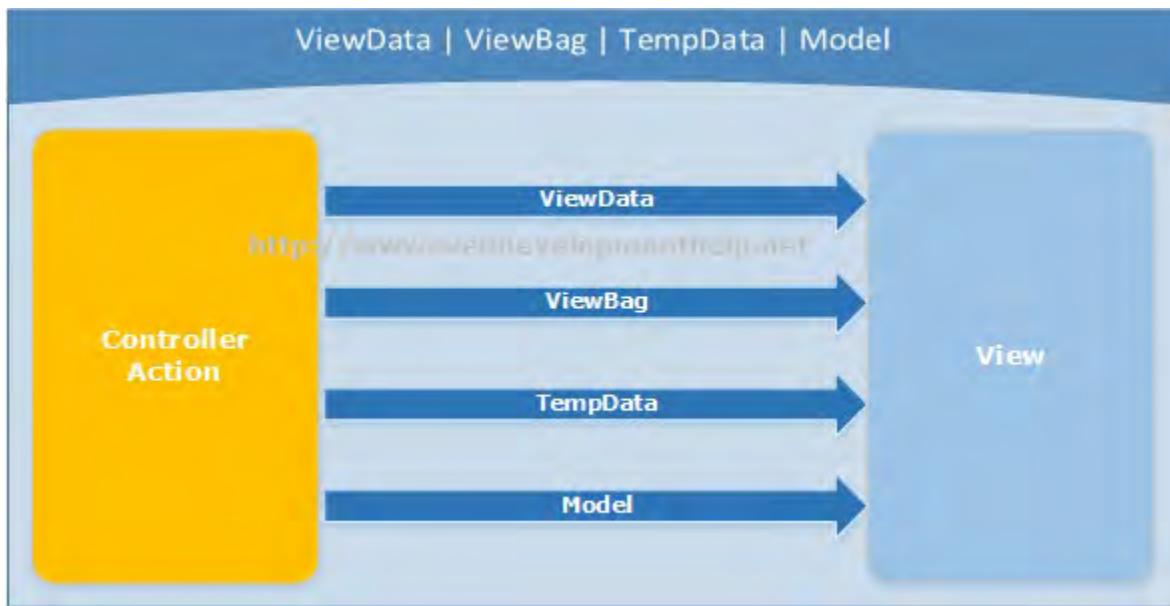
Passing data in MVC

ASP.NET MVC is a framework that facilitates building web applications based on MVC (Model-View-Controller) design pattern. Request coming from client reaches the *Controller* through URL Rewriting Module. *Controller* decides which model to use in order to fulfill the request. Further passing the *Model* data to *View* which then transforms the *Model* data and renders response to client as shown in following basic level request flow diagram.



In this ASP.NET MVC Tutorial, we will discuss and implement different options to pass data from ASP.NET MVC *Controller* to *View*. Following are the different available options to pass data from a *Controller* to *View* in ASP.NET MVC along with Introduction and Background:

- Introduction and Background
- ViewBag
- ViewData
- TempData (TempData Vs Session)
- Model



Introduction

If we want to maintain state between a *Controller* and corresponding *View*- ***ViewData*** and ***ViewBag*** are the available options but both of these options are limited to a single server call (meaning it's value will be null if a redirect occurs). But if we need to maintain state from one *Controller* to another (redirect case), then ***TempData*** is the other available option.

It's common that initially it might be a bit difficult for a ASP.NET WebForms developer to digest above flow and need for options to pass data from *Controller* to *View*. Because in WebForms approach, Controller and View are tightly coupled to each other. Please follow the link for a detailed comparison of the [differences between ASP.NET WebForms and ASP.NET MVC](#) here.

For the purpose of implementation, we will take earlier [ASP.NET MVC tutorial](#) on this blog as base and implement with different options. If you haven't gone through the article, please read "[Building your first ASP.NET MVC application in 4 simple steps](#)" first.

ViewBag Example

As we discussed earlier that *ViewBag* and *ViewData* serves the same purpose but *ViewBag* is basically a dynamic property (a new C# 4.0 feature) having advantage that it doesn't have typecasting and null checks.

So, In order to pass data from Controller to View using *ViewBag*, we will modify our EmployeeController code as follows:

```
public class EmployeeController : Controller
{
    // GET: /Employee/
```

```

public ActionResult Index()
{
    ViewBag.EmployeeName = "Muhammad Hamza";
    ViewBag.Company = "Web Development Company";
    ViewBag.Address = "Dubai, United Arab Emirates";
    return View();
}
}

```

And to get Employee details passed from Controller using ViewBag, View code will be as follows:

```

<body>
<div>
    <h1>Employee (ViewBag Data Example)</h1>
    <div>
        <b>Employee Name:</b> @ViewBag.EmployeeName<br />
        <b>Company Name:</b> @ViewBag.Company<br />
        <b>Address:</b> @ViewBag.Address<br />
    </div>
</div>
</body>

```

In order to see the above changes in action run the solution, we will find the following output.

Employee (ViewBag Data Example)

Employee Name: Muhammad Hamza
Company Name: Web Development Company
Address: Dubai, United Arab Emirates

ViewData Example

As compared to ViewBag, ViewData is a dictionary object which requires typecasting as well as null checks. Same above implementation using ViewData can be achieved as follows:

```

public class EmployeeController : Controller
{
    // GET: /Employee/
    public ActionResult Index()
    {

```

```

        ViewData["EmployeeName"] = "Muhammad Hamza";
        ViewData["Company"] = "Web Development Company";
        ViewData["Address"] = "Dubai, United Arab Emirates";

        return View();
    }
}

```

And to get Employee details passed from Controller using ViewBag, View code will be as follows:

```

<body>
<div>
<h1>Employee (ViewBag Data Example)</h1>
<div>
<b>Employee Name:</b> @ViewData["EmployeeName"]<br />
<b>Company Name:</b> @ViewData["Company"]<br />
<b>Address:</b> @ViewData["Address"]<br />
</div>
</div>
</body>

```

Run the application to view the following output.



Employee (ViewData Example)

Employee Name: Muhammad Hamza
Company Name: Web Development Company
Address: Dubai, United Arab Emirates

TempData

TempData in ASP.NET MVC is basically a dictionary object derived from TempDataDictionary. TempData stays for a subsequent HTTP Request as opposed to other options (ViewBag and ViewData) those stay only for current request. So, TempData can be used to maintain data between controller actions as well as redirects.

Note: Just like ViewData, typecasting and null checks required for TempData also in order to avoid errors.

Let's see how we can use TempData in a practical scenario to pass data from one controller action to another.

```
//Controller Action 1 (TemporaryEmployee)
public ActionResult TemporaryEmployee()
{
    Employee employee = new Employee
    {
        EmpID = "121",
        EmpFirstName = "Imran",
        EmpLastName = "Ghani"
    };
    TempData["Employee"] = employee;
    return RedirectToAction("PermanentEmployee");
}

//Controller Action 2(PermanentEmployee)
public ActionResult PermanentEmployee()
{
    Employee employee = TempData["Employee"] as Employee;
    return View(employee);
}
```

As in above example, we store an employee object in TempData in Controller Action 1 (i.e. TemporaryEmployee) and retrieve it in another Controller Action 2 (i.e. PermanentEmployee). But If we try to do the same using ViewBag or ViewData, we will get null in Controller Action 2 because only TempData object maintains data between controller actions.

Model

Now in this part, we are going to implement another approach i.e. using Model for passing data from ASP.NET MVC Controller to View. As we already know that in MVC (Model, View, Controller) pattern, Model represents our domain model corresponding to tables in database. So, we can use this model class to serve the purpose.

Consider the following Model class we have already used in our previous implementations:

```
namespace MyMVCAApp.Models
{
    public class Employee
    {
        public string EmpID { get; set; }
        public string EmpFirstName { get; set; }
```

```

        public string EmpLastName { get; set; }
    }
}

```

Now, we are done with our Model class (i.e. Employee). For the purpose of this implementation, we are loading data directly to our Model in EmployeeController. In a real implementation, we must be fetching this data from a data source e.g. a database. So, the Controller has following code:

```

public class EmployeeController : Controller
{
    public ActionResult Index()
    {
        List<Employee> employees = new List<Employee>()
        {
            new Employee{EmpID = "1", EmpFirstName = "Imran", EmpLastName = "Ghani"},
            new Employee{EmpID = "2", EmpFirstName = "Rizwan", EmpLastName = "Mukhtar"},
            new Employee{EmpID = "3", EmpFirstName = "Rehan", EmpLastName = "Ahmad"},
            new Employee{EmpID = "4", EmpFirstName = "Zeeshan", EmpLastName = "Khalid"},
            new Employee{EmpID = "5", EmpFirstName = "Sajid", EmpLastName = "Majeed"}
        };
        return View(employees);
    }
}

```

In above code example, you can see that as opposite to ViewData and ViewBag, we are passing data (i.e. employees) as parameter to View instead of placing in another store and calling as View().

Finally, updating our View to get and display data passed from Controller as:

```

@model IEnumerable<MyMVCAccount.Models.Employee>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Employee Index Page</title>
</head>
<body>
    <div>
        <h1>Employee (Using Model)</h1>
        <div>

```

```

@foreach(var employee in Model){
    <p>@employee.EmpID @employee.EmpFirstName @employee.EmpLastName</p>
>
    }
</div>
</div>
</body>
</html>

```

At the top of our View (i.e. Index.cshtml), we have referred to our Model class (i.e. Employee) using IEnumerable. Further in View, we are using foreach to access and display data as needed. When we run this application, our View will display the Model data from Controller as follows:



Using Model to pass data to View in ASP.NET is pretty simple as well as more appropriate because we are fetching data from permanent store and loading in our Model classes in most of the cases. Also, we use it to perform all CRUD (Create, Retrieve, Update, Delete) operations. Hopefully, this ASP.NET MVC Tutorial will help in understanding the mentioned approach in a more effective way.

The article is written by Web Development Tutorial

Tutorial: Create a web API with ASP.NET Core

Article • 12/07/2021 • 46 minutes to read •  +35

[Is this page helpful?](#)

In this article

[Overview](#)

[Prerequisites](#)

[Create a web project](#)

[Add a model class](#)

[Add a database context](#)

[Add the TodoContext database context](#)

[Register the database context](#)

[Scaffold a controller](#)

[Examine the PostTodoItem create method](#)

[Examine the GET methods](#)

[Routing and URL paths](#)

[Return values](#)

[The PutTodoItem method](#)

[The DeleteTodoItem method](#)

[Prevent over-posting](#)

[Call the web API with JavaScript](#)

[Add authentication support to a web API](#)

[Additional resources](#)

By [Rick Anderson](#) and [Kirk Larkin](#)

This tutorial teaches the basics of building a web API with ASP.NET Core.

In this tutorial, you learn how to:

- ✓ Create a web API project.
- ✓ Add a model class and a database context.
- ✓ Scaffold a controller with CRUD methods.
- ✓ Configure routing, URL paths, and return values.

✓ Call the web API with Postman.

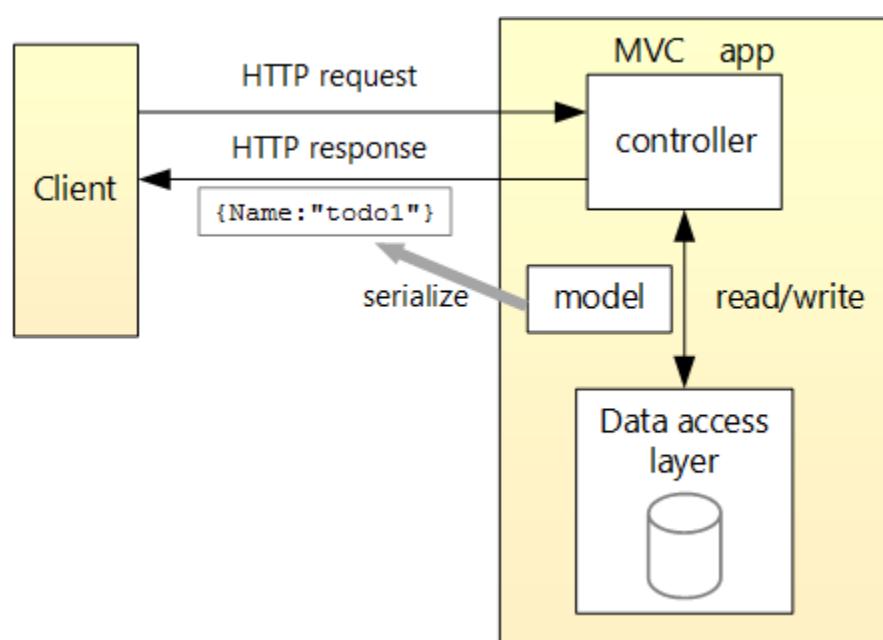
At the end, you have a web API that can manage "to-do" items stored in a database.

Overview

This tutorial creates the following API:

API	Description	Request body	Response body
GET /api/todoitems	Get all to-do items	None	Array of to-do items
GET /api/todoitems/{id}	Get an item by ID	None	To-do item
POST /api/todoitems	Add a new item	To-do item	To-do item
PUT /api/todoitems/{id}	Update an existing item	To-do item	None
DELETE /api/todoitems/{id}	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites

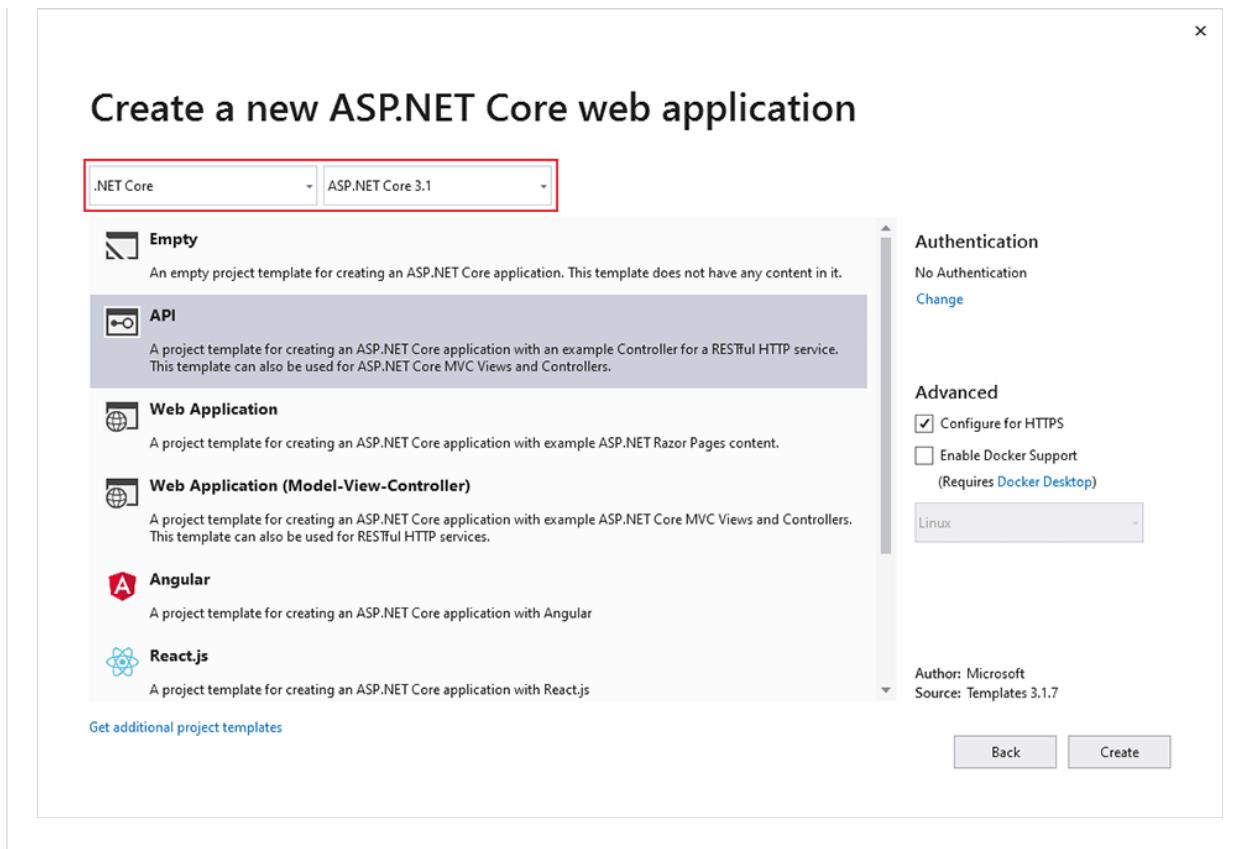
[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

- [Visual Studio 2019 16.4 or later](#) with the **ASP.NET and web development** workload
- [.NET Core 3.1 SDK](#)

Create a web project

[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

- From the **File** menu, select **New > Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 3.1** are selected. Select the **API** template and click **Create**.



Test the API

The project template creates a `WeatherForecast` API. Call the `Get` method from a browser to test the app.

Visual Studio Visual Studio Code Visual Studio for Mac

Press **Ctrl+F5** to run the app. Visual Studio launches a browser and navigates to `https://localhost:<port>/WeatherForecast`, where `<port>` is a randomly chosen port number.

If you get a dialog box that asks if you should trust the IIS Express certificate, select **Yes**. In the **Security Warning** dialog that appears next, select **Yes**.

JSON similar to the following is returned:

```
JSON Copy
[  
 {  
   "date": "2019-07-16T19:04:05.7257911-06:00",  
 }
```

```
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

[Visual Studio](#)[Visual Studio Code](#)[Visual Studio for Mac](#)

- In **Solution Explorer**, right-click the project. Select **Add > New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add > Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following code:

C#

 Copy

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

Add a database context

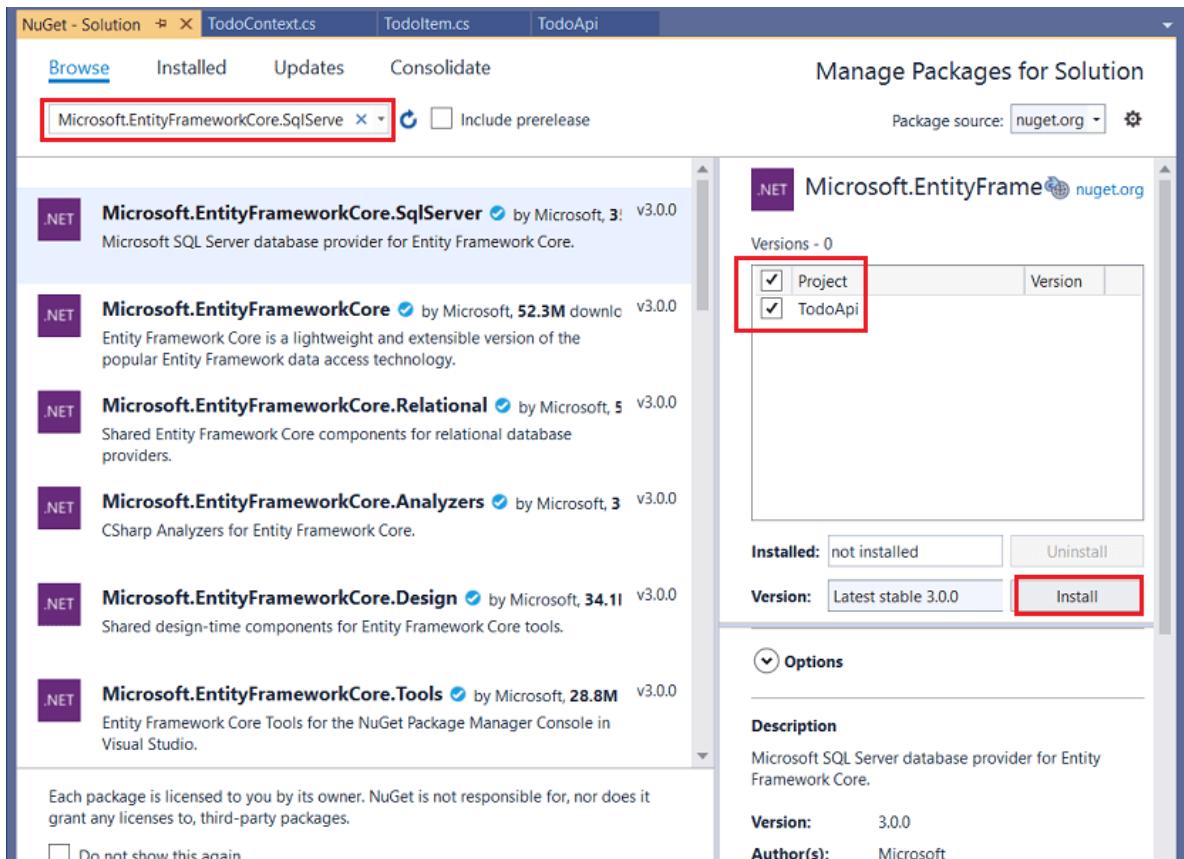
The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

[Visual Studio](#)[Visual Studio Code / Visual Studio for Mac](#)

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter `Microsoft.EntityFrameworkCore.InMemory` in the search box.
- Select `Microsoft.EntityFrameworkCore.InMemory` in the left pane.

- Select the **Project** checkbox in the right pane and then select **Install**.



Add the TodoContext database context

- Right-click the *Models* folder and select **Add > Class**. Name the class *TodoContext* and click **Add**.

- Enter the following code:

```
C#  
Copy  
  
using Microsoft.EntityFrameworkCore;  
  
namespace TodoApi.Models  
{  
    public class TodoContext : DbContext  
    {  
        public TodoContext(DbContextOptions<TodoContext> options)  
            : base(options)  
        {  
        }  
    }  
}
```

```
        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the [dependency injection \(DI\)](#) container. The container provides the service to controllers.

Update *Startup.cs* with the following highlighted code:

C#

 Copy

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
```

```
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

The preceding code:

- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

Visual Studio

Visual Studio Code / Visual Studio for Mac

- Right-click the `Controllers` folder.
- Select **Add > New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
 - Select **TodoItem (TodoApi.Models)** in the **Model class**.
 - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
 - Select **Add**.

The generated code:

- Marks the class with the [\[ApiController\]](#) attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see [Create web APIs with ASP.NET Core](#).
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the [CRUD](#) methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the `action` name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

Examine the `PostTodoItem` create method

Replace the return statement in the `PostTodoItem` to use the `nameof` operator:

C#	 Copy
<pre>// POST: api/TodoItems [HttpPost] public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem) { _context.TodoItems.Add(todoItem); await _context.SaveChangesAsync(); //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem); return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem); }</pre>	

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute.

The method gets the value of the to-do item from the body of the HTTP request.

For more information, see [Attribute routing with `Http\[Verb\]` attributes](#).

The `CreatedAtAction` method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.

- Adds a [Location](#) header to the response. The `Location` header specifies the `URI` of the newly created to-do item. For more information, see [10.2.2 201 Created](#).
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Install Postman

This tutorial uses Postman to test the web API.

- [Install Postman](#)
- Start the web app.
- Start Postman.
- Disable **SSL certificate verification**
 - From **File > Settings (General tab)**, disable SSL certificate verification.

 **Warning**

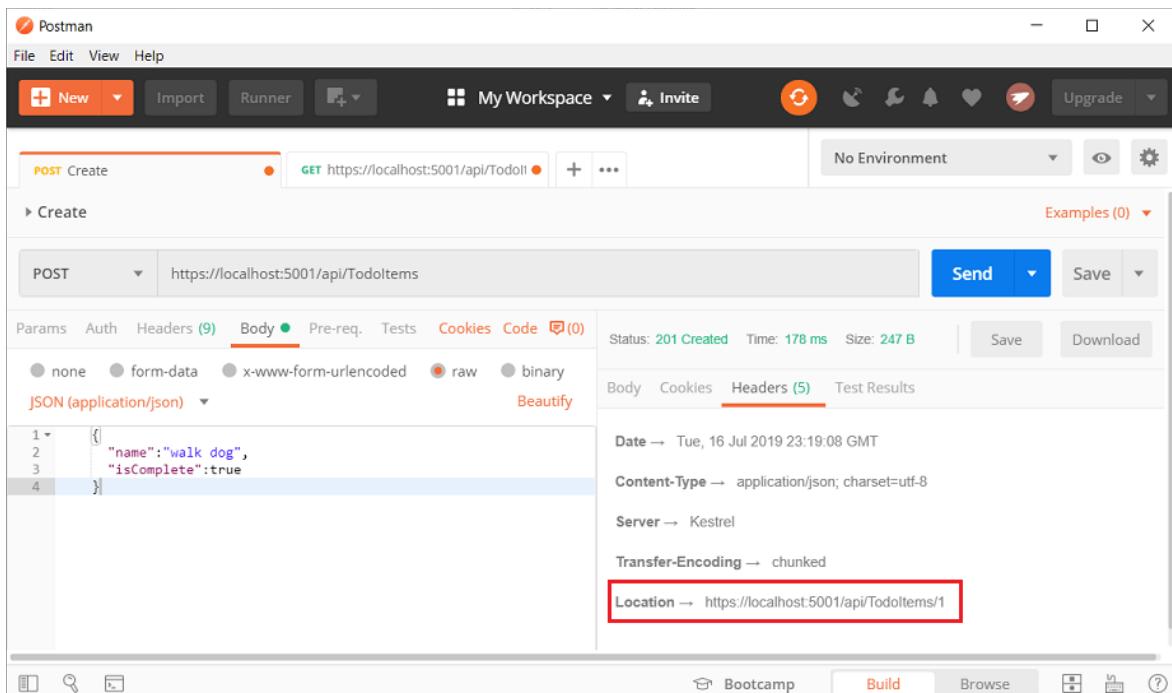
Re-enable SSL certificate verification after testing the controller.

Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/todoitems`. For example, `https://localhost:5001/api/todoitems`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

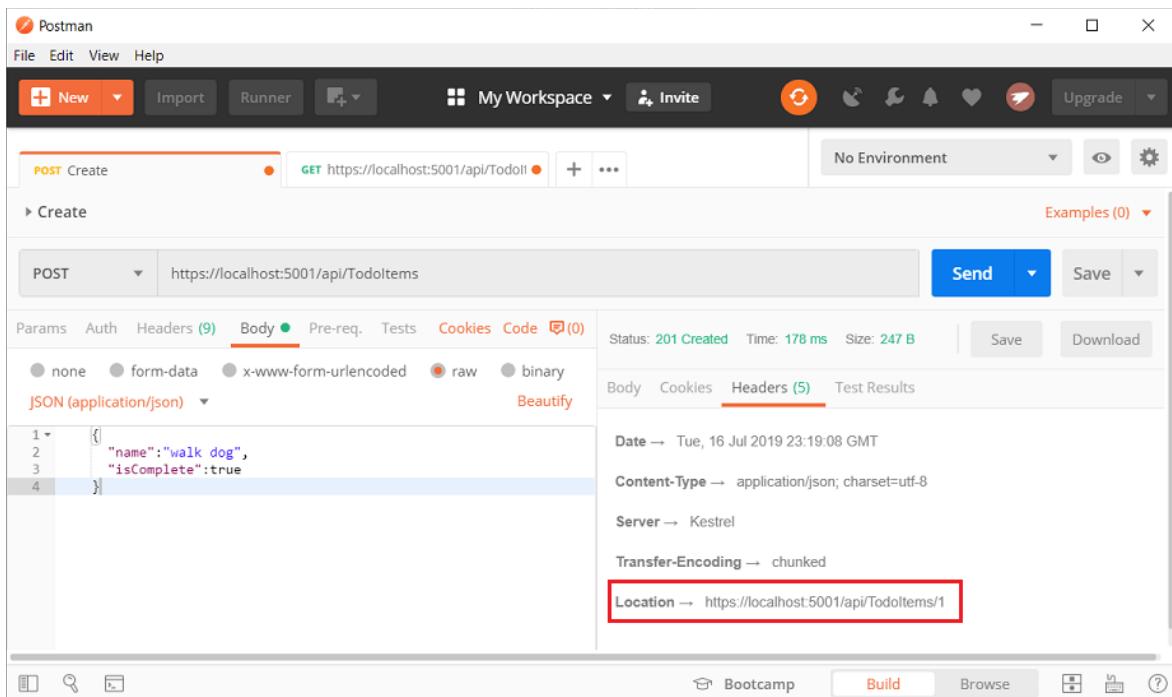
```
JSON Copy  
  
{  
    "name": "walk dog",  
    "isComplete": true  
}
```

- Select **Send**.



Test the location header URI with Postman

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the HTTP method to `GET`.
- Set the URI to `https://localhost:<port>/api/todoitems/1`. For example, `https://localhost:5001/api/todoitems/1`.
- Select **Send**.

Examine the GET methods

These methods implement two GET endpoints:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

Test the app by calling the two endpoints from a browser or Postman. For example:

- `https://localhost:5001/api/todoitems`
- `https://localhost:5001/api/todoitems/1`

A response similar to the following is produced by the call to `GetTodoItems`:

```
JSON Copy  
[  
 {
```

```
[{"id": 1,  
 "name": "Item1",  
 "isComplete": false  
}]
```

Test Get with Postman

- Create a new request.
- Set the HTTP method to **GET**.
- Set the request URI to `https://localhost:<port>/api/todoitems`. For example, `https://localhost:5001/api/todoitems`.
- Set **Two pane view** in Postman.
- Select **Send**.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, **POST** data to the app.

Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

C#	 Copy
<pre>[Route("api/[controller]")] [ApiController] public class TodoItemsController : ControllerBase { private readonly TodoContext _context; public TodoItemsController(TodoContext context) { _context = context; } }</pre>	

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller

class name is `TodoItemsController`, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.

- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see [Attribute routing with `Http\[Verb\]` attributes](#).

In the following `GetTodoItem` method, "`{id}`" is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of "`{id}`" in the URL is provided to the method in its `id` parameter.

```
C#  
  
// GET: api/TodoItems/5  
[HttpGet("{id}")]  
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)  
{  
    var todoItem = await _context.TodoItems.FindAsync(id);  
  
    if (todoItem == null)  
    {  
        return NotFound();  
    }  
  
    return todoItem;  
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is [ActionResult<T> type](#). ASP.NET Core automatically serializes the object to [JSON](#) and writes the JSON into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 [NotFound](#) error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

 Copy

```
// PUT: api/TodoItems/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is [204 \(No Content\)](#). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use [HTTP PATCH](#).

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

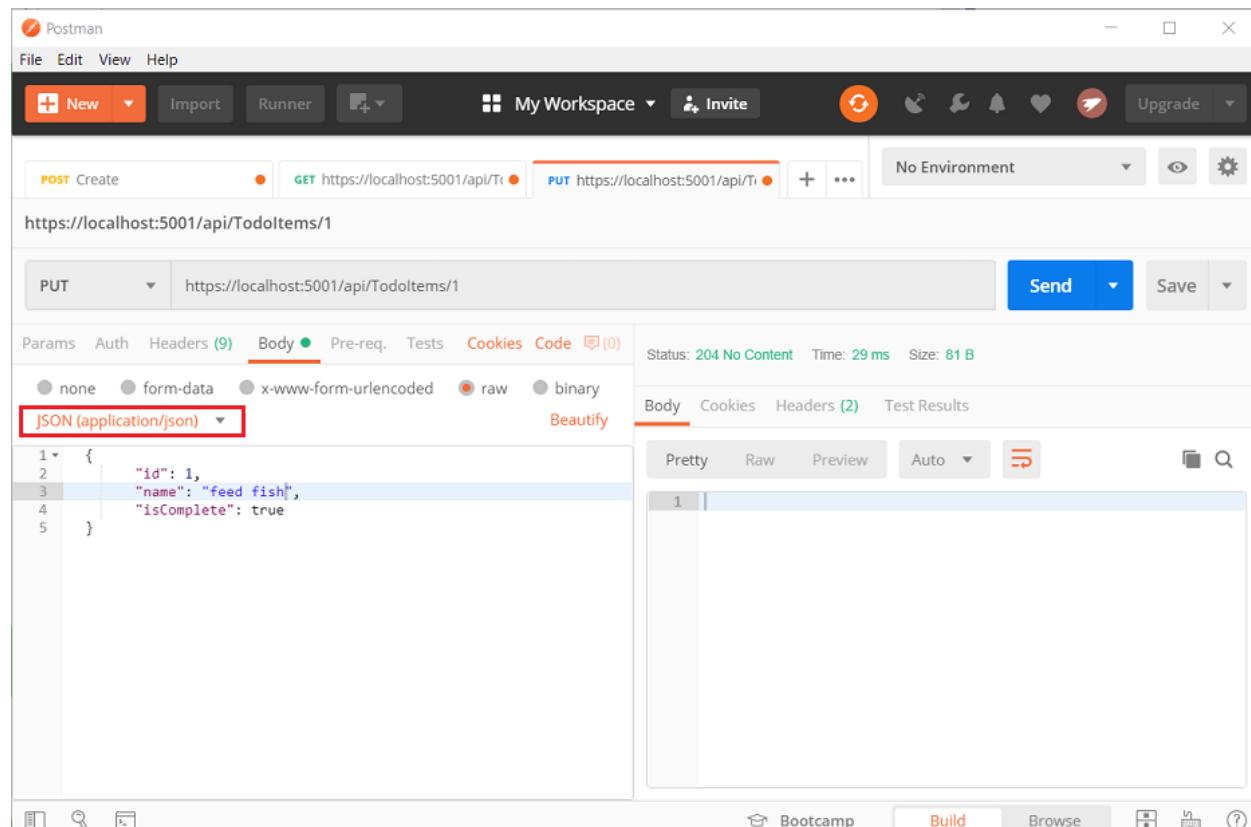
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
JSON Copy  
  
{  
    "id":1,  
    "name":"feed fish",  
    "isComplete":true  
}
```

The following image shows the Postman update:



The screenshot shows the Postman interface. The top navigation bar includes File, Edit, View, Help, and a My Workspace dropdown. Below the header, there are three tabs: POST Create, GET https://localhost:5001/api/TodoItems/1, and PUT https://localhost:5001/api/TodoItems/1. The PUT tab is selected. The URL https://localhost:5001/api/TodoItems/1 is entered in the request URL field. The method is set to PUT. In the Body section, the Content Type is set to "application/json". The JSON payload is displayed in a code editor:

```
1 {  
2     "id": 1,  
3     "name": "feed fish",  
4     "isComplete": true  
5 }
```

The status bar at the bottom indicates a 204 No Content response with a size of 81 B.

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

 Copy

```
// DELETE: api/TodoItems/5
[HttpDelete("{id}")]
public async Task<ActionResult<TodoItem>> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return todoItem;
}
```

Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example `https://localhost:5001/api/todoitems/1`).
- Select **Send**.

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be

more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

C#

 Copy

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
    public string Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

C#

 Copy

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

 Copy

```
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
{
    return await _context.TodoItems
        .Select(x => ItemToDTO(x))
        .ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
```

```
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO
todoItemDTO)
    {
        if (id != todoItemDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoItemDTO.Name;
        todoItem.IsComplete = todoItemDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }

    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO
todoItemDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoItemDTO.IsComplete,
            Name = todoItemDTO.Name
        };
    }
}
```

```
_context.TodoItems.Add(todoItem);
await _context.SaveChangesAsync();

return CreatedAtAction(
    nameof(GetTodoItem),
    new { id = todoItem.Id },
    ItemToDTO(todoItem));
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}

private bool TodoItemExists(long id) =>
    _context.TodoItems.Any(e => e.Id == id);

private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
    new TodoItemDTO
    {
        Id = todoItem.Id,
        Name = todoItem.Name,
        IsComplete = todoItem.IsComplete
    };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See [Tutorial: Call an ASP.NET Core web API with JavaScript](#).

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- [Azure Active Directory](#)
- [Azure Active Directory B2C](#) (Azure AD B2C)
- [IdentityServer4](#)

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see [Welcome to IdentityServer4](#).

Additional resources

[View or download sample code for this tutorial](#). See [how to download](#).

For more information, see the following resources:

- [Create web APIs with ASP.NET Core](#)
- [ASP.NET Core web API documentation with Swagger / OpenAPI](#)
- [Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8](#)
- [Routing to controller actions in ASP.NET Core](#)
- [Controller action return types in ASP.NET Core web API](#)
- [Deploy ASP.NET Core apps to Azure App Service](#)
- [Host and deploy ASP.NET Core](#)
- [Microsoft Learn: Create a web API with ASP.NET Core](#)



JavaScript basics

JavaScript is a programming language that adds interactivity to your website. This happens in games, in the behavior of responses when buttons are pressed or with data entry on forms; with dynamic styling; with animation, etc. This article helps you get started with JavaScript and furthers your understanding of what is possible.

What is JavaScript?

[JavaScript](#) is a powerful programming language that can add interactivity to a website. It was invented by Brendan Eich (co-founder of the Mozilla project, the Mozilla Foundation, and the Mozilla Corporation).

JavaScript is versatile and beginner-friendly. With more experience, you'll be able to create games, animated 2D and 3D graphics, comprehensive database-driven apps, and much more!

JavaScript itself is relatively compact, yet very flexible. Developers have written a variety of tools on top of the core JavaScript language, unlocking a vast amount of functionality with minimum effort. These include:

- Browser Application Programming Interfaces ([APIs](#)) built into web browsers, providing functionality such as dynamically creating HTML and setting CSS styles; collecting and manipulating a video stream from a user's webcam, or generating 3D graphics and audio samples.
- Third-party APIs that allow developers to incorporate functionality in sites from other content providers, such as Twitter or Facebook.
- Third-party frameworks and libraries that you can apply to HTML to accelerate the work of building sites and applications.

It's outside the scope of this article—as a light introduction to JavaScript—to present the details of how the core JavaScript language is different from the tools listed above. You can learn more in MDN's [JavaScript learning area](#), as well as in other parts of MDN.

The section below introduces some aspects of the core language and offers an opportunity to play with a few browser API features too. Have fun!

A *Hello world!* example

JavaScript is one of the most popular modern web technologies! As your JavaScript skills grow,

the standard for introductory programming examples .)

Warning: If you haven't been following along with the rest of our course, [download this example code](#) and use it as a starting point.

1. Go to your test site and create a new folder named `scripts`. Within the `scripts` folder, create a new file called `main.js`, and save it.

2. In your `index.html` file, enter this code on a new line, just before the closing `</body>` tag:

```
<script src="scripts/main.js"></script>
```



3. This is doing the same job as the [`<link>`](#) element for CSS. It applies the JavaScript to the page, so it can have an effect on the HTML (along with the CSS, and anything else on the page).

4. Add this code to the `main.js` file:

```
const myHeading = document.querySelector('h1');  
myHeading.textContent = 'Hello world!';
```



5. Make sure the HTML and JavaScript files are saved. Then load `index.html` in your browser. You should see something like this:



Note: The reason the instructions (above) place the `<script>` element near the bottom of the HTML file is that **the browser reads code in the order it appears in the file**.

If the JavaScript loads first and it is supposed to affect the HTML that hasn't loaded yet, there could be problems. Placing JavaScript near the bottom of an HTML page is one way

Following that, the code set the value of the `myHeading` variable's [textContent](#) property (which represents the content of the heading) to *Hello world!*.

Note: Both of the features you used in this exercise are parts of the [Document Object Model \(DOM\) API](#), which has the capability to manipulate documents.

Language basics crash course

To give you a better understanding of how JavaScript works, let's explain some of the core features of the language. It's worth noting that these features are common to all programming languages. If you master these fundamentals, you have a head start on coding in other languages too!

Warning: In this article, try entering the example code lines into your JavaScript console to see what happens. For more details on JavaScript consoles, see [Discover browser developer tools](#).

Variabiles

[Variables](#) are containers that store values. You start by declaring a variable with the [var](#) (less recommended, dive deeper for the explanation) or the [let](#) keyword, followed by the name you give to the variable:

```
let myVariable;
```



Note: A semicolon at the end of a line indicates where a statement ends. It is only required when you need to separate statements on a single line. However, some people believe it's

between var and let.

After declaring a variable, you can give it a value:

```
myVariable = 'Bob';
```



Also, you can do both these operations on the same line:

```
let myVariable = 'Bob';
```



You retrieve the value by calling the variable name:

```
myVariable;
```



After assigning a value to a variable, you can change it later in the code:

```
let myVariable = 'Bob';
myVariable = 'Steve';
```

Note that variables may hold values that have different [data types](#):

Variable	Explanation	Example
	This is a	

	don't need quote marks.	
<u>Array</u>	This is a structure that allows you to store multiple values in a single reference.	<pre>let myVariable = [1, 'Bob', 'Steve', 10];</pre> Refer to each member of the array like this: <code>myVariable[0]</code> , <code>myVariable[1]</code> , etc.
<u>Object</u>	This can be anything. Everything in JavaScript is an object and can be stored in a variable. Keep this in mind as you learn.	<pre>let myVariable = document.querySelector('h1');</pre> All of the above examples too.

Addition	Add two numbers together or combine two strings.	+	<code>6 + 9; 'Hello ' + 'world!';</code>
Subtraction, Multiplication, Division	These do what you'd expect them to do in basic math.	- , * , /	<code>9 - 3; 8 * 2; // multiply in JS is an asterisk 9 / 3;</code>
Assignment	As you've seen already: this assigns a	=	<code>let myVariable = 'Bob';</code>

There are a lot more operators to explore, but this is enough for now. See [Expressions and operators](#) for a complete list.

Note: Mixing data types can lead to some strange results when performing calculations. Be careful that you are referring to your variables correctly, and getting the results you expect. For example, enter `'35' + '25'` into your console. Why don't you get the result you expected? Because the quote marks turn the numbers into strings, so you've ended up

—it is likely a function. Functions often take [arguments](#): bits of data they need to do their job. Arguments go inside the parentheses, separated by commas if there is more than one argument.

For example, the `alert()` function makes a pop-up box appear inside the browser window,



How to call a JSON API and display the result in ASP.NET Core MVC

In today's world of interconnected apps & services, you may often need to call another API to get or modify information.

We are going to do the following:

- call another API in a reliable fashion
- receive and parse the result so that your app can work with it
- display the result on a web page

Setting your project up

This tutorial uses ASP.NET Core 3 MVC. If you are using a different version, some of the steps may need to be modified.

If you are starting from scratch, you can just create a new ASP.NET Core MVC project by running the following command:

```
dotnet new mvc -o MyMvcApp
```

where `MyMvcApp` is the name for your new project. If you already have a project, ignore this step.

Warning! The step below is only going to work if your project is using .NET Core 2.2 or a higher version. If you're on a lower version, you really should upgrade. Alternatively, just create a new project that runs on the latest .NET Core

Given your app will be calling some remote API, you want to make sure that in case of a flaky connection or a timeout it's going to re-try to eventually get the data. That's why you going to need to add a reference to `Microsoft.Extensions.Http.Polly` package. To read more about Polly and re-try policies, check out this article: "[Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies](#)"

So, to add this package, run this command from your project folder (that's where your `.csproj` resides):

```
dotnet add package Microsoft.Extensions.Http.Polly
```

Also, you'll need a `Newtonsoft.Json` package so that you could turn all this JSON into neat C# objects that are easier to work with. So go ahead and run this command to add it:

```
dotnet add package newtonsoft.json
```

Haaang on, wait, but why do I need all this?

There are several approaches and classes you may use to call external APIs, and you might have come across some of them already - classes like `WebClient` or `HttpClient`. While still working, they provide quite manual and low-level access to APIs. However in web apps, since they may have several threads running and calling different APIs, you want to let the application manage creation, re-use and disposal of these resources.

That's why one of the recommended practices at the moment is to use `HttpClientFactory` pattern. That is a standard part of .NET Core, and it takes good care of things like configuring `HttpClient`s, managing creation and disposal of them (to avoid socket exhaustion, that's when OS runs out of available network sockets) and does a few more things for you – see [here](#) for the complete list.

Also, feel free to check out all the use cases of the `HttpClientFactory` [here](#) – the class is easy to use and examples are quite clear.

You will also need `Polly` (the package you just installed above) to add retry policies.

Configuring your web app

Add the following code to the `Startup.cs` file:

```
// Add this line to your 'using' section:
```

```

using Polly;

//
// ...existing code already there...
//
    // Add this code into your 'ConfigureServices' function:
    public void ConfigureServices(IServiceCollection services)
    {
        // Existing code, most likely you already have it there
        services.AddControllersWithViews();

        // Now let's register an API client for your AJAX call.
        // Includes the configuration - base address & content type.
        services.AddHttpClient("API Client", client => {
            client.BaseAddress = new Uri("https://www.metaweather.com/");
            client.DefaultRequestHeaders.Add("Accept", "application/json");
        })
        // Add the re-try policy: in this instance, re-try three times,
        // in 1, 3 and 5 seconds intervals.
        .AddTransientHttpErrorPolicy(builder => builder.WaitAndRetryAsync(new[] {
            TimeSpan.FromSeconds(1),
            TimeSpan.FromSeconds(5),
            TimeSpan.FromSeconds(10)
        }));
    }
}

```

In this example we are going to use weather forecasting API from [MetaWeather](#) - it's free and doesn't need any API keys or other credentials, so should be easy to get working on your computer.

Call all the APIs!

OMG can't believe we actually got here after all this setup! Bear with me, we're really close, soon you'll be getting that sweet JSON and displaying it!

First of all, you'll need to create a class (or set of classes) to represent your JSON structures in C#. Luckily, there's a web app for that! Head to <http://json2csharp.com/>, paste your expected JSON response in there, click 'Generate' and you'll get your C# class(es).

We are going to use [this MetaWeather API response](#) to generate C# classes. Feel free to change the generated class names, however, do not change the field names, as those need to match your JSON.

In my case, I get something like this:

```

public class WeatherForecast
{
    public List<ConsolidatedWeather> consolidated_weather { get; set; }
    public DateTime time { get; set; }
    public DateTime sun_rise { get; set; }
    public DateTime sun_set { get; set; }
    public string timezone_name { get; set; }
    public Parent parent { get; set; }
    public List<Source> sources { get; set; }
    public string title { get; set; }
    public string location_type { get; set; }
    public int woeid { get; set; }
    public string latt_long { get; set; }
    public string timezone { get; set; }
}

```

Now it's time to call the API, and turn the response into an instance of that freshly created class. In one of your controllers, add the following code:

```

// Reference Newtonsoft
using Newtonsoft.Json;

// ... other code...

private async Task<WeatherForecast> GetWeatherForecasts()
{
    // Get an instance of HttpClient from the factory that we registered
    // in Startup.cs
    var client = _httpClientFactory.CreateClient("API Client");

    // Call the API & wait for response.
    // If the API call fails, call it again according to the re-try policy
    // specified in Startup.cs
    var result = await client.GetAsync("/api/location/1103816/");

    if (result.IsSuccessStatusCode)
    {
        // Read all of the response and deserialise it into an instance of
        // WeatherForecast class
        var content = await result.Content.ReadAsStringAsync();
        return JsonConvert.DeserializeObject<WeatherForecast>(content);
    }
    return null;
}

public async Task<IActionResult> Index()
{
    var model = await GetWeatherForecasts();
    // Pass the data into the View
    return View(model);
}

```

Things to pay particular attention to:

- Notice how we get an instance `HttpClient` with the help of `HttpClientFactory`
- `result.IsSuccessStatusCode` needs to be checked before proceeding to retrieve the response
- `JsonConvert.DeserializeObject<WeatherForecast>()` call does all the magic of converting JSON response in string form to C# class instance

Showing the result

Now is the easy bit – you already have your strongly-typed model, representing the JSON result, and you can render it to the user:

```
@model WeatherForecast;
<div class="text-left">
    <h1 class="display-4">Weather forecast for @Model.title</h1>
    @foreach (var forecast in Model.consolidated_weather)
    {
        <div>
            <h4>@forecast.applicable_date</h4>
            <div>Min: @forecast.min_temp C°, Max: @forecast.max_temp C°</div>
            <div>@forecast.weather_state_name</div>
            
            <hr/>
        </div>
    }
</div>
```

Which gives us something like this:

Weather forecast for Melbourne

2019-10-14

Min: 12.175 C°, Max: 20.215 C°

Light Rain



2019-10-15

Min: 9.645 C°, Max: 18.23 C°

Light Cloud



2019-10-16

Min: 8.87 C°, Max: 16.265 C°

Heavy Rain



Conclusion

In this codelab, you have just learnt how to:

- Set up your project to call remote APIs in a reliable, production-ready manner
- Generated C# class(es) to contain data from JSON API that you are calling
- Call the API & deserialise the result into instances of C# class(es)
- Display the result on the page

Code lab 8

 Hide Folder Information

Instructions

Code Lab 8

Assigned: Wednesday 12/1

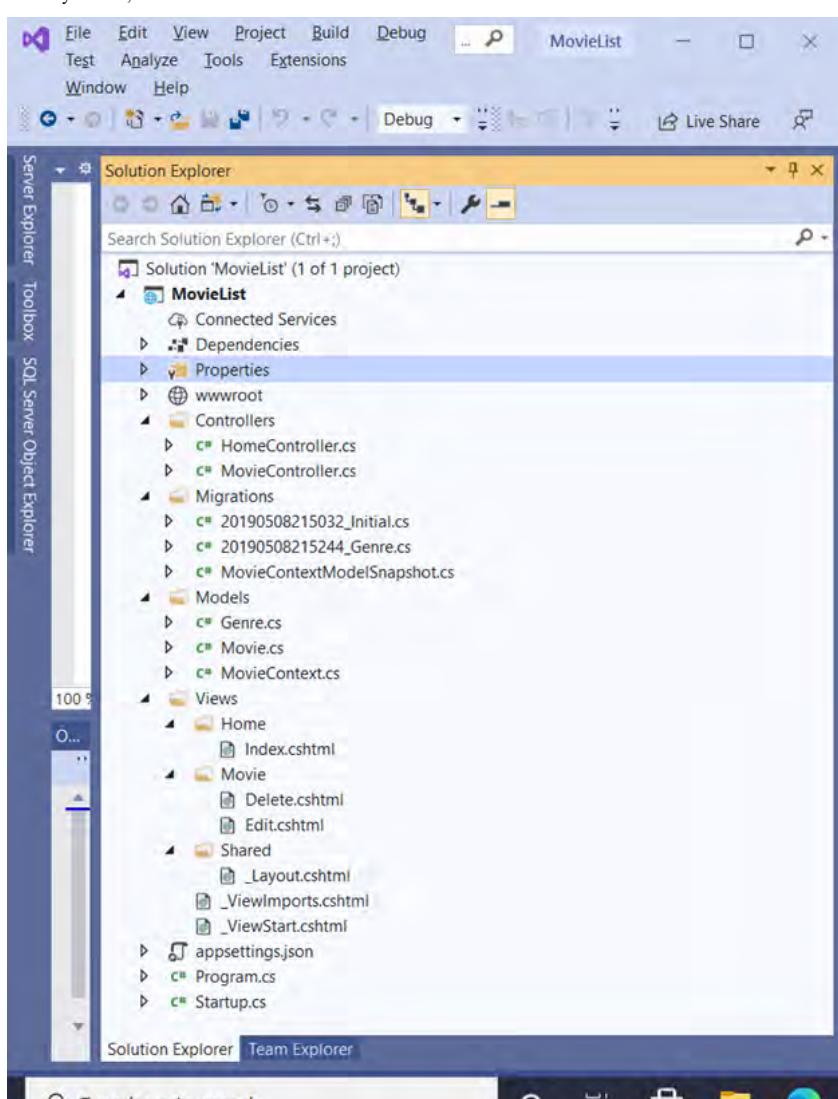
You should complete the code lab 8 by Wednesday, December 8

This code lab guides you through the development of the Movie List app. This will give you a chance to generate a database from entity classes.

Set up the file structure

1. Create a new ASP.NET Core Web Application with a project name of MovieList and a solution name of MovieList. Base this app on the Web Application (Model-View-Controller) template.

2. Using the figure below as a guide, remove all unnecessary files such as Models/ ErrorViewModel.cs, Views/Home/Privacy.cshtml, and so on.



3. Using the procedure below as a guide, install the EF Core and EF Core Tools NuGet packages.

How to open the NuGet Package Manager

- Select ToolsàNuget Package ManageràManage NuGet Packages for Solution. The NuGet Package Manager

How to install the EF Core and EF Core Tools NuGet packages

1. Click the Browse link in the upper left of the window.
 2. Type "Microsoft.EntityFrameworkCore.SqlServer" in the search box.
 3. Click on the appropriate package from the list that appears in the left-hand panel.
 4. In the right-hand panel, check the project name, select the version that matches the version of .NET Core you're running, and click Install.
 5. Review the Preview Changes dialog that comes up and click OK.
 6. Review the License Acceptance dialog that comes up and click I Accept.
 7. Type "Microsoft.EntityFrameworkCore.Tools" in the search box.
 8. Repeat steps 3 through 6. Description
- With .NET Core 3.0 and later, you must manually add EF Core and EF Core Tools to your project.

4. Using the procedure below as a guide, use LibMan to install version 4.3.1 of the Bootstrap CSS library. In the Solution Explorer, expand the wwwroot/lib folder and make a note of the path to the bootstrap.min.css file.

1. Start Visual Studio and open a project. In the Solution Explorer, expand the wwwroot/lib folder and delete any old Bootstrap or jQuery libraries.
2. In the Solution Explorer, right-click on the project name and select the AddàClient-Side Library item.
3. In the dialog box that appears, type "jquery@", select "3.3.1" from the list that appears, and click the Install button.
4. Repeat steps 2 and 3 for the library named "twitter-bootstrap@4.3.1", but change the target location to "www/lib/bootstrap".
5. Repeat steps 2 and 3 for the library named "popper.js@1.14.7".
6. Repeat steps 2 and 3 for the library named "jquery-validate@1.19.0".
7. Repeat steps 2 and 3 for the library named "jquery-validation-unobtrusive@3.2.11".
8. In the Solution Explorer, expand the wwwroot/lib folder and view the libraries that have been installed. If you didn't already do it in step 4, rename the twitter-bootstrap folder to bootstrap.

Note:

You can use the Library Manager, also known as LibMan, to add client-side libraries such as Bootstrap and jQuery to a project.

Modifying the existing file

5. Modify the Views/Shared/_Layout.cshtml file so it contains this code:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" type="text/css" href="~/lib/bootstrap/dist/css/bootstrap.min.css">
</head>
<body>
    <div class="container">
        <header class="jumbotron">
            <h1>My Movies</h1>
        </header>
        @RenderBody()
    </div>
</body>
</html>
```

Make sure the href attribute specifies the correct path to the bootstrap.min.css file!

6. Remove all action methods from the HomeController except for the Index() method.

Code the classes for the model and the DB context

7. Add a Movie class to the Models folder and edit it so it contains the code shown below. Don't forget to add the using directive for data annotations.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MovieList.Models
{
    public class Movie
    {
        // EF will instruct the database to automatically generate this value
```

```

public int MovieId { get; set; }

[Required(ErrorMessage = "Please enter a name.")]
public string Name { get; set; }

[Required(ErrorMessage = "Please enter a year.")]
[Range(1889, 2050, ErrorMessage = "Year must be between 1889 and now.")]
public int? Year { get; set; }

[Required(ErrorMessage = "Please enter a rating.")]
[Range(1, 5, ErrorMessage = "Rating must be between 1 and 5.")]
public int? Rating { get; set; }
}
}

```

8. Add a MovieContext class to the Models folder and edit it so it contains the code shown below. Don't forget to add the using directive for the EF Core namespace.

```

using Microsoft.EntityFrameworkCore;
namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {}

        public DbSet<Movie> Movies { get; set; }
        public DbSet<Genre> Genres { get; set; }
    }
}

```

Note:

DbContext The primary class for communicating with a database.

DbContextOptions Stores configuration options for the DbContext object.

DbSet A collection of objects created from the specified entity

- Within the DbContext class, you can use DbSet properties to work with the model classes that map to database tables.

These classes are also known as entity classes, or domain model classes.

- Any property in your entity with a name of Id (or ID) or the entity name followed by Id (or ID) is a primary key. If this property is also of the int type, the corresponding column is an identity column whose value is automatically generated.
9. Modify the MovieContext class to contain the code for the OnModelCreating() method shown below.

```

namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {}

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Movie>().HasData(
                new Movie {
                    MovieId = 4,
                    Name = "Casablanca",
                    Year = 1943,
                    Rating = 5,
                },
                new Movie {
                    MovieId = 2,
                    Name = "Wonder Woman",
                    Year = 2017,
                    Rating = 3,
                },
                new Movie {

```

```

        MovieId = 3,
        Name = "Moonstruck",
        Year = 1988,
        Rating = 4,
    }
}

}
}

```

10. Add the connection string to appsettings.json as shown below. Specify a name of MoviesExercise for the database by editing the Database parameter like this: Database=Movies Make sure to enter the entire connection string on one line and to add a comma to the end of the previous line.

```

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

```

11. Modify the Startup.cs file so it includes the code shown below. This enables dependency injection for DbContext objects. At the top of the file, make sure to include all of the necessary using directives, including the using directive for the Models namespace.

```

using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using MovieList.Models;
namespace MovieList
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRouting(options => {
                options.LowercaseUrls = true;
                options.AppendTrailingSlash = true;
            });
            services.AddControllersWithViews();
            services.AddDbContext<MovieContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
        }
    }
}

```

12. Remove any statements from the Startup.cs file that aren't needed by the Movie List app. Your Startup.cs file should look like the one below.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

```

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using MovieListPN.Models;
namespace MovieListPN
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        public IConfiguration Configuration { get; }
        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
            services.AddDbContext<MovieContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
        }
        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms
                /aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseAuthorization();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}

```

Create the Movies database

13. Using the procedure below as a guide, open the Package Manager Console. At the command prompt, enter the "Add-Migration Initial" command. This should add a Migrations folder and migration files to the Solution Explorer. If you get an error, troubleshoot the problem.

How to open the Package Manager Console window

- Select the Tools NuGet Package Manager Package Manager Console command.

How to create the Movies database based on your code files

1. Make sure the connection string and dependency injection are set up.
2. Type "Add-Migration Initial" in the PMC at the command prompt and press Enter.

14. At the Package Manager Console command prompt, enter the "Update-Database" command. This should create the database. If you get an error, troubleshoot the problem.

Note: You should see code below in the Up() method of the intial migration file.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Movies",
        columns: table => new
        {
            Movield = table.Column<int>(type: "int", nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Year = table.Column<int>(type: "int", nullable: false),
            Rating = table.Column<int>(type: "int", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Movies", x => x.Movield);
        });
    migrationBuilder.InsertData(
        table: "Movies",
        columns: new[] { "Movield", "Name", "Rating", "Year" },
        values: new object[] { 4, "Casablanca", 5, 1943 });
    migrationBuilder.InsertData(
        table: "Movies",
        columns: new[] { "Movield", "Name", "Rating", "Year" },
        values: new object[] { 2, "Wonder Woman", 3, 2017 });
    migrationBuilder.InsertData(
        table: "Movies",
        columns: new[] { "Movield", "Name", "Rating", "Year" },
        values: new object[] { 3, "Moonstruck", 4, 1988 });
}
```

15. View your database. To do that, display the SQL Server Object Explorer as described in figure 4-7. Then, expand the nodes until you can see the Movies database that you just created.

1. Choose the View  SQL Server Object Explorer command in Visual Studio.
2. Expand the (localdb)\MSSQLLocalDB node, then expand the Databases node.
3. Expand the Movies node, then expand the Tables node.
4. To view the table columns, expand a table node and then its Columns node.
5. To view the table data, right-click a table and select the ViewData command

16. View the seed data. To do that, expand the Movies node and the Tables node. Then, right-click on the dbo.Movies table and select View Data. This should show the data that's stored in the Movies table.

Modify the Home controller and its view

17. Modify the Home controller so it contains the code shown below. At the top of the controller, make sure to include using directives for all necessary namespaces including the Models namespace.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using MovieList.Models;
namespace MovieListPN.Controllers
{
    public class HomeController : Controller
    {
        private MovieContext context { get; set; }
        public HomeController(MovieContext ctx)
        {
            context = ctx;
        }
    }
}
```

```

public IActionResult Index()
{
    var movies = context.Movies.OrderBy(m => m.Name).ToList();
    return View(movies);
}

}

```

18. Modify the Home/Index view so it contains the code shown below.

```

@model List<Movie>
 @{
     ViewData["Title"] = "Home Page";
 }
 <h2>Movie List PN</h2>
 <a asp-controller="Movie" asp-action="Add">Add New Movie</a>
 <table class="table table-bordered table-striped">
     <thead>
         <tr><th>Name</th><th>Year</th><th>Rating</th><th></th></tr>
     </thead>
     <tbody>
         @foreach (var movie in Model)
         {
             <tr>
                 <td>@movie.Name</td>
                 <td>@movie.Year</td>
                 <td>@movie.Rating</td>
                 <td>
                     <a asp-controller="Movie" asp-action="Edit"
                         asp-route-id="@movie.MovieId">Edit</a>
                     <a asp-controller="Movie" asp-action="Delete"
                         asp-route-id="@movie.MovieId">Delete</a>
                 </td>
             </tr>
         }
     </tbody>
 </table>

```

Add the Movie controller and its views

20. Add a new controller named MovieController to the Controllers folder. Then, modify this controller so it contains the code shown below. Again, make sure to include the using directive for the Models namespace.

```

namespace MovieListPN.Controllers
{
    public class MovieController : Controller
    {
        private MovieContext context { get; set; }
        public MovieController(MovieContext ctx)
        {
            context = ctx;
        }
        [HttpGet]
        public IActionResult Add()
        {
            ViewBag.Action = "Add";
            return View("Edit", new Movie());
        }
        [HttpGet]
        public IActionResult Edit(int id)
        {
            ViewBag.Action = "Edit";
            var movie = context.Movies.Find(id);
            return View(movie);
        }
    }
}

```

```
        }
        [HttpPost]
        public IActionResult Edit(Movie movie)
        {
            if (ModelState.IsValid)
            {
                if (movie.MovieId == 0)
                    context.Movies.Add(movie);
                else
                    context.Movies.Update(movie);
                context.SaveChanges();
                return RedirectToAction("Index", "Home");
            }
            else
            {
                ViewBag.Action = (movie.MovieId == 0) ? "Add" : "Edit";
                return View(movie);
            }
        }
        [HttpGet]
        public IActionResult Delete(int id)
        {
            var movie = context.Movies.Find(id);
            return View(movie);
        }
        [HttpPost]
        public IActionResult Delete(Movie movie)
        {
            context.Movies.Remove(movie);
            context.SaveChanges();
            return RedirectToAction("Index", "Home");
        }
    }
}
```

21. Add a folder named Movie under the Views folder.

22. Add a new view named Edit to the Views/Movie folder (Select Razor View Empty). Then, modify this view so it contains the code shown below.

```
@model Movie
 @{
    string title = ViewBag.Action + " Movie";
    ViewBag.Title = title;
}
<h2>@title</h2>
<form asp-action="Edit" method="post">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Name">Name</label>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Year">Year</label>
        <input asp-for="Year" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Rating">Rating</label>
        <input asp-for="Rating" class="form-control" />
    </div>
    <input type="hidden" asp-for="MovieId" />
    <button type="submit" class="btn btn-primary">@ViewBag.Action</button>
    <a asp-controller="Home" asp-action="Index" class="btn btn-primary">Cancel</a>
</form>
```

23. Add a new view named Delete to the Views/Movie folder. Then, modify this view so it contains the code shown below.

```
@model Movie
 @{
    ViewBag.Title = "Delete Movie";
}
<h2>Confirm Deletion</h2>
<h3>@Model.Name (@Model.Year)</h3>
<form asp-action="Delete" method="post">
    <input type="hidden" asp-for="MovieId" />
    <button type="submit" class="btn btn-primary">Delete</button>
    <a asp-controller="Home" asp-action="Index" class="btn btn-primary">Cancel</a>
</form>
```

24. Run the app. It should display the list of movies. In addition, you should be able to add, edit, and delete movies.

Start Date

Dec 1, 2021 1:00 PM

Due Date

Dec 14, 2021 11:00 AM

Submit Assignment

Files to submit *

(0) file(s) to submit

After uploading, you must click Submit to complete the submission.

Add a File

Record Audio

Comments

Submit

Cancel

HTML Helpers in ASP.NET MVC

An HTML Helper is just a method that returns a HTML string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML <input>, <button> and tags etc.

You can also create your own HTML Helpers to render more complex content such as a menu strip or an HTML table for displaying database data.

Different types of HTML Helpers

There are three types of HTML helpers as given below:

Inline Html Helpers

These are created in the same view by using the Razor @helper tag. These helpers can be reused only on the same view.

```
1. @helper ListingItems(string[] items)
2. {
3.     <ol>
4.     @foreach (string item in items)
5.     {
6.         <li>@item</li>
7.     }
8.     </ol>
9. }
10.
11.    <h3>Programming Languages:</h3>
12.
13.    @ListingItems(new string[] { "C", "C++", "C#" })
14.
15.    <h3>Book List:</h3>
16.
17.    @ListingItems(new string[] { "How to C", "how to C++", "how to C#" })
```

Built-In Html Helpers

Built-In Html Helpers are extension methods on the `HtmlHelper` class. The Built-In Html helpers can be divided into three categories.

Standard Html Helpers

These helpers are used to render the most common types of HTML elements like as HTML text boxes, checkboxes etc. A list of most common standard html helpers is given below:

HTML Element

TextBox

```
@Html.TextBox("Textbox1", "val")
Output: <input id="Textbox1" name="Textbox1" type="text" value="val" />
```

TextArea

```
@Html.TextArea("Textarea1", "val", 5, 15, null)
Output: <textarea cols="15" id="Textarea1" name="Textarea1" rows="5">val</textarea>
```

Password

```
@Html.Password("Password1", "val")
Output: <input id="Password1" name="Password1" type="password" value="val" />
```

Hidden Field

```
@Html.Hidden("Hidden1", "val")
Output: <input id="Hidden1" name="Hidden1" type="hidden" value="val" />
```

CheckBox

```
@Html.CheckBox("Checkbox1", false)
Output: <input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" /> <input name="myCheckbox" type="hidden" value="false" />
```

RadioButton

```
@Html.RadioButton("Radiobutton1", "val", true)
Output: <input checked="checked" id="Radiobutton1" name="Radiobutton1" type="radio" value="val" />
```

Drop-down list

```
@Html.DropDownList ("DropDownList1", new SelectList(new [] {"Male", "Female"}))
Output: <select id="DropDownList1" name="DropDownList1"> <option>M</option>
<option>F</option> </select>
```

Multiple-select

```
Html.ListBox("ListBox1", new MultiSelectList(new [] {"Cricket", "Chess"}))
Output: <select id="ListBox1" multiple="multiple" name="ListBox1">
<option>Cricket</option> <option>Chess</option> </select>
```

Strongly Typed HTML Helpers

These helpers are used to render the most common types of HTML elements in strongly typed view like as HTML text boxes, checkboxes etc. The HTML elements are created based on model properties.

The strongly typed HTML helpers work on lambda expression. The model object is passed as a value to lambda expression, and you can select the field or property from model object to be used to set the id, name and value attributes of the HTML helper. A list of most common strongly-typed html helpers is given below:

HTML Element

TextBox

```
@Html.TextBoxFor(m=>m.Name)  
Output: <input id="Name" name="Name" type="text" value="Name-val" />
```

TextArea

```
@Html.TextArea(m=>m.Address , 5, 15, new{})  
Output: <textarea cols="15" id="Address" name="Address"  
rows="5">Address value</textarea>
```

Password

```
@Html.PasswordFor(m=>m.Password)  
Output: <input id="Password" name="Password" type="password"/>
```

Hidden Field

```
@Html.HiddenFor(m=>m.UserId)  
Output: <input id="UserId" name="UserId" type="hidden" value="UserId-val" />
```

CheckBox

```
@Html.CheckBoxFor(m=>m.IsApproved)  
Output: <input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" /> <input  
name="myCheckbox" type="hidden" value="false" />
```

RadioButton

```
@Html.RadioButtonFor(m=>m.IsApproved, "val")  
Output: <input checked="checked" id="RadioButton1" name="RadioButton1" type="radio"  
value="val" />
```

Drop-down list

```
@Html.DropDownListFor(m => m.Gender, new SelectList(new [] {"Male", "Female"}))  
Output: <select id="Gender" name="Gender"> <option>Male</option>  
<option>Female</option> </select>
```

Multiple-select

```
Html.ListBoxFor(m => m.Hobbies, new MultiSelectList(new [] {"Cricket", "Chess"}))  
Output: <select id="Hobbies" multiple="multiple" name="Hobbies">  
<option>Cricket</option> <option>Chess</option> </select>
```

Templated HTML Helpers

These helpers figure out what HTML elements are required to render based on properties of your model class. This is a very flexible approach for displaying data to the user, although it requires some initial care and attention to set up. To setup proper HTML element with Templated HTML Helper, make use of `DataType` attribute of `DataAnnotation` class.

For example, when you use `DataType` as `Password`, A templated helper automatically render `Password` type HTML input element.

Templated Helper

Display

Renders a read-only view of the specified model property and selects an appropriate HTML element based on property's data type and metadata.

```
Html.Display("Name")
```

DisplayFor

Strongly typed version of the previous helper

```
Html.DisplayFor(m => m.Name)
```

Editor

Renders an editor for the specified model property and selects an appropriate HTML element based on property's data type and metadata.

```
Html.Editor("Name")
```

EditorFor

Strongly typed version of the previous helper

```
Html.EditorFor(m => m.Name)
```

Custom Html Helpers

You can also create your own custom helper methods by creating an extension method on the `HtmlHelper` class or by creating static methods with in a utility class.

HTML Helpers and Creating Custom HTML Helpers in ASP.NET MVC

People coming from the asp.net web forms background are used to putting the ASP.NET server control on the page using the toolbox. When we work with ASP.NET MVC application there is no toolbox available to us from where we can drag and drop HTML controls on the view. In MVC, if we want to create a view it should contain HTML code for specifying the mark up. MVC Beginners(specially with Web forms background) finds this a little troubling.

ASP.NET MVC team must have anticipated this problem and thus to ease this problem, the ASP.NET MVC framework comes with a set of HTML Helper methods. These helpers are simple functions that let the developer to specify the type of HTML needed on the view. This is done in C#. The final HTML will be generated by these functions at the runtime i.e. We don't have to worry about the correctness of generated HTML.

Built in HTML Helpers

Let's look at how we can use various HTML helper methods. Lets try to create a simple contact us form that will ask the user for his name, email id and his query. we can design this form in simple HTML easily, let us see how we can utilize HTML helper to achieve the same.

Contact.cshtml

Client Objects & Events	(No Events)
-------------------------	-------------

```
<h2>Contact</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>ContactInfo</legend>

        <div class="editor-label">
            @Html.Label("Name")
        </div>
        <div class="editor-field">
            @Html.Editor("txtName")
        </div>

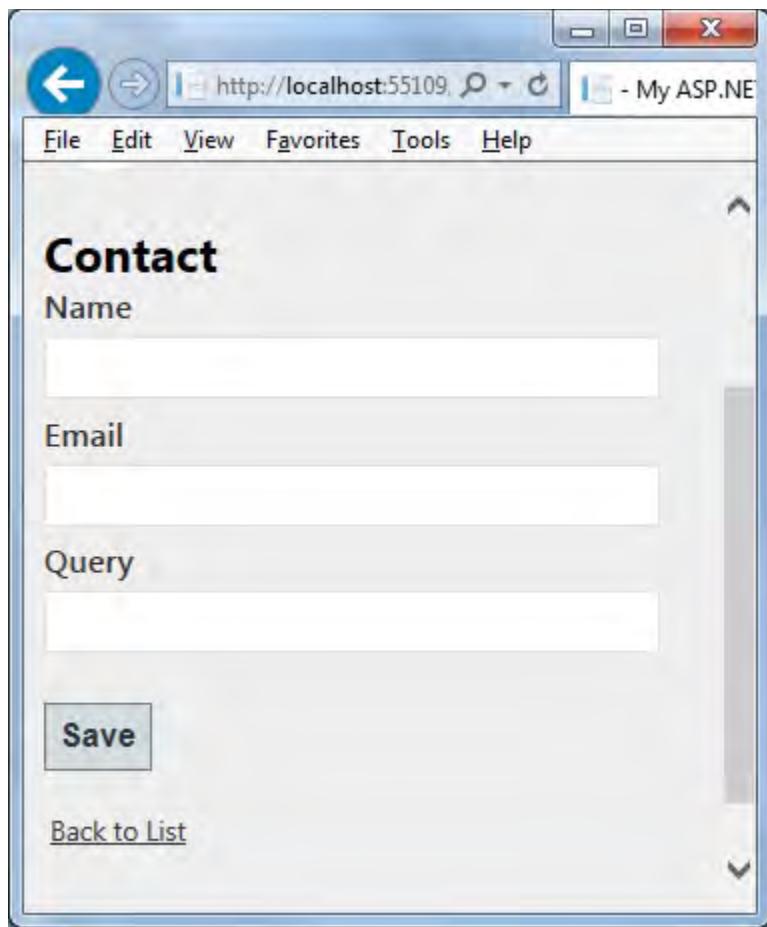
        <div class="editor-label">
            @Html.Label("Email")
        </div>
        <div class="editor-field">
            @Html.Editor("txtEmail")
        </div>

        <div class="editor-label">
            @Html.Label("Query")
        </div>
        <div class="editor-field">
            @Html.Editor("txtQuery")
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

In the above screenshot we can see that the HTML form is created using a HTML helper function, all the labels on the page are created using helper functions and all the textboxes are also created using helper functions. Now if we run the application and try to see the result:



If we want to control the HTML attributes of the generated HTML then we can use the overloaded version of the helper functions. Lets say in the above code, we want to provide `ID` to all the labels.

Contact.cshtml

Client Objects & Events	(No Events)
-------------------------	-------------

```
<h2>Contact</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>ContactInfo</legend>

        <div class="editor-label">
            @Html.Label("Name", new { id = "lblName" })
        </div>
        <div class="editor-field">
            @Html.Editor("txtName")
        </div>

        <div class="editor-label">
            @Html.Label("Email", new { id = "lblEmail" })
        </div>
        <div class="editor-field">
            @Html.Editor("txtEmail")
        </div>

        <div class="editor-label">
            @Html.Label("Query", new { id = "lblQuery" })
        </div>
        <div class="editor-field">
            @Html.Editor("txtQuery")
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

If we run the application again and try to look at the generated HTML, we can see the ID values we have specified.



```
File Edit Format
31         </div>
32     </header>
33     <div id="body">
34
35         <section class="content-wrapper main-content clear-
fix">
36             <h2>Contact</h2>
37
38 <form action="/Home/Contact" method="post">      <fieldset>
39     <legend>ContactInfo</legend>
40
41     <div class="editor-label">
42         <label for="Name" id="lblName">Name</label>
43     </div>
44     <div class="editor-field">
45         <input class="text-box single-line" id="txtName"
name="txtName" type="text" value="" />
46     </div>
47
48     <div class="editor-label">
49         <label for="Email" id="lblEmail">Email</label>
50     </div>
51     <div class="editor-field">
```

Following HTML helpers are built into the ASP.NET MVC framework:

- `Html.BeginForm`
- `Html.EndForm`
- `Html.TextBox`
- `Html.TextArea`
- `Html.Password`
- `Html.Hidden`
- `Html.CheckBox`
- `Html.RadioButton`
- `Html.DropDownList`
- `Html.ListBox`

HTML Helpers for strongly typed views

If we are creating a strongly typed view then it is also possible to use the HTML helpers methods with the model class. Let us create a `model` for the contact us page:

Hide Copy Code

```
public class ContactInfo
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Query { get; set; }
}
```

Now let's create a strongly typed view for contact us page using the Html helpers.

```
Client Objects & Events (No Events)
@model HTMLHelpersDemo.Models.ContactInfo
<h2>Contact</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <div class="editor-label">
        @Html.LabelFor(model => model.Name)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Name)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Email)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Email)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Query)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Query)
    </div>

    <p>
        <input type="submit" value="Save" />
    </p>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

These helper methods create the output HTML elements based on model properties. The property to be used to create the HTML is passed to the method as a lambda expression. It could also be possible to specify id, name and various other HTML attributes using these helper methods. Following HTML helpers are available to be used with strongly typed views:

- `Html.TextBoxFor`
- `Html.TextAreaFor`
- `Html.PasswordFor`

- `Html.HiddenFor`
- `Html.CheckBoxFor`
- `Html.RadioButtonFor`
- `Html.DropDownListFor`
- `Html.ListBoxFor`

Creating Custom HTML Helpers

If we want to create our own HTML helpers than that can also be done very easily. There are quite a few ways of creating custom helpers. lets look at all the possible methods.

1. Creating a static method
2. Writing an extension method
3. Using the `@helper(razor only)`

Let us try to create a simple HTML helper method which will create a marked HTML label. There is a new mark tag in HTML5 specifications. Let us try to create a label which create a label then mark it using the HTML5 mark tag.

Let's try to achieve this using all the above mentioned methods.

Creating a static method

In this approach we can simply create a static class with static method which will return the HTML string to be used at the time of rendering.

```
namespace HTMLHelpersDemo.Helpers
{
    public static class MyHTMLHelpers
    {
        public static IHtmlString LabelWithMark(string content)
        {
            string htmlString =
String.Format("<label><mark>{0}</mark></label>", content);
            return new HtmlString(htmlString);
        }
    }
}
```

Writing an extension method

In this approach we will write a simple extension method for the built in html helper class. this will enable us to use the same Html object to use our custom helper method.

```
public static class MyExtensionMethods
{
    public static IHtmlString LabelWithMark(this HtmlHelper helper, string
content)
    {
```

```
        string htmlString = String.Format("<label><mark>{0}</mark></label>",  
content);  
        return new HtmlString(htmlString);  
    }  
}
```

Using the @helper(razor only)

This method is pretty specific to razor view engine. Let us see how this can be done. We need to write this method in the view itself.

[Hide](#) [Copy Code](#)

```
@helper LabelWithMarkRazor(string content)  
{  
    <label><mark>@content</mark></label>  
}
```

Note: By default these helpers will be available in the view they are defined in. If we want to use the razor created helpers in multiple views then we have to put all of them in one view and put that view file in `App_Code` directory. This way they will be usable from all the views.

Now we have the same html helper method written in 3 different ways, Let us create a simple view and try to use these helper methods.

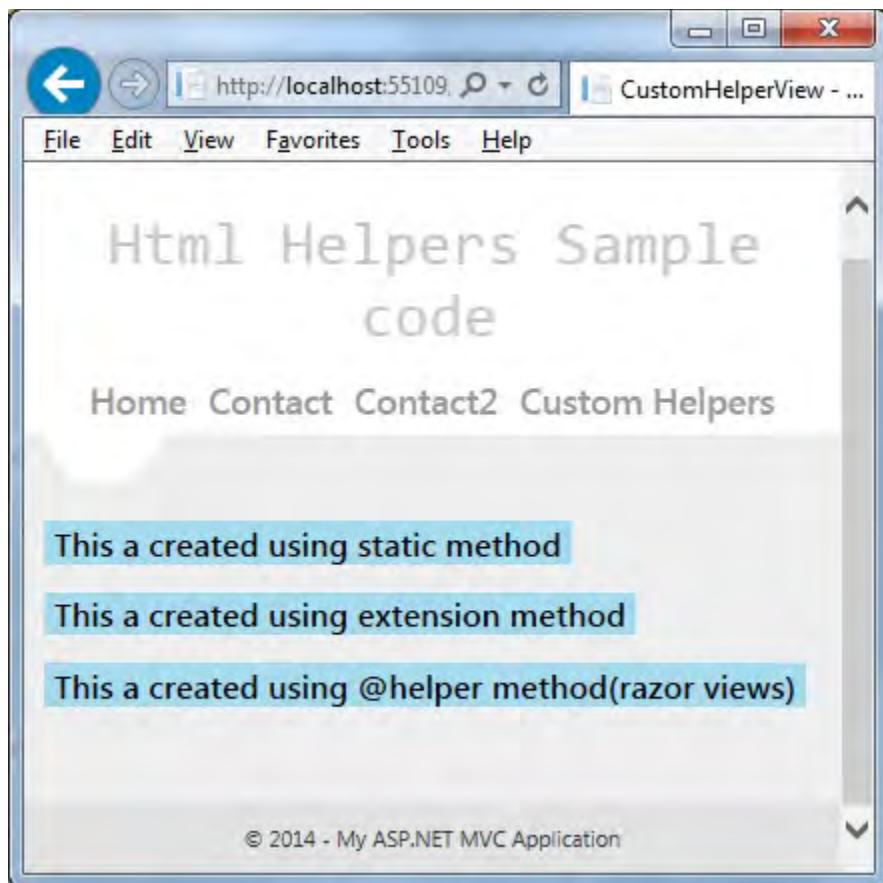
CustomHelperView.cshtml

Client Objects & Events (No Events)

```
@{  
    ViewBag.Title = "CustomHelperView";  
}  
  
@using HTMLHelpersDemo.Helpers  
  
- <p>  
  @MyHTMLHelpers.LabelWithMark("This a created using static method")  
  </p>  
  
- <p>  
  @Html.LabelWithMark("This a created using extension method")  
  </p>  
  
@helper LabelWithMarkRazor(string content)  
{  
    <label><mark>@content</mark></label>  
}  
  
- <p>  
  @LabelWithMarkRazor("This a created using @helper method(razor views)")  
  </p>
```

100 %

Let's run the application and see the result:



In this article we looked at HTML helpers. We have also looked at how we can create custom HTML helpers.

Written by **Rahul Rajat Singh**

How to develop a data-driven MVC web app

Objectives (part 1)

Applied

1. Given the specifications for a multi-page web app that stores data in a database, code and test the app.
2. Use Visual Studio to add the NuGet packages for EF Core and its tools to a project.

Knowledge

1. Describe how you can code a primary key for an entity by convention and describe when the value for that primary key will be automatically generated.
2. Describe how a DB context class maps related entity classes to a database and seeds the database with initial data.
3. Name the file that's typically used to store a connection string for a database.

Objectives (part 2)

4. Describe how a Startup.cs file can read a connection string from a file.
5. Describe how ASP.NET Core uses dependency injection to pass DbContext objects to the controllers that need them.
6. Describe how to use the Package Manager Console to create migration files and use them to create and update the database of a web app.
7. Describe how to use LINQ to query the data that's available from the DbSet properties of a DbContext object.
8. Describe how to use the methods of the DbSet and DbContext classes to add, update, or delete entities in the database.
9. Describe how you can code a foreign key for an entity by convention.

Objectives (part 3)

10. Describe how to use the Startup.cs file to modify the HTTP request and response pipeline to provide for user-friendly URLs.
11. Describe how slugs can make URLs more user friendly.

The Movie List page of the Movie List app

A screenshot of a web browser window titled "My Movies". The address bar shows the URL <https://localhost:5001>. The main content area displays the heading "My Movies" and "Movie List". Below this, there is a table with three rows, each representing a movie entry. The table has columns for Name, Year, Genre, Rating, and actions (Edit and Delete). The movies listed are Casablanca (1942, Drama, Rating 5), Moonstruck (1988, RomCom, Rating 4), and Wonder Woman (2017, Action, Rating 4).

Name	Year	Genre	Rating	
Casablanca	1942	Drama	5	Edit Delete
Moonstruck	1988	RomCom	4	Edit Delete
Wonder Woman	2017	Action	4	Edit Delete

The Add Movie page of the Movie List app

A screenshot of a web browser window titled "Add Movie". The URL in the address bar is "https://localhost:5001/movie/add". The page content is as follows:

My Movies

Add Movie

Name:

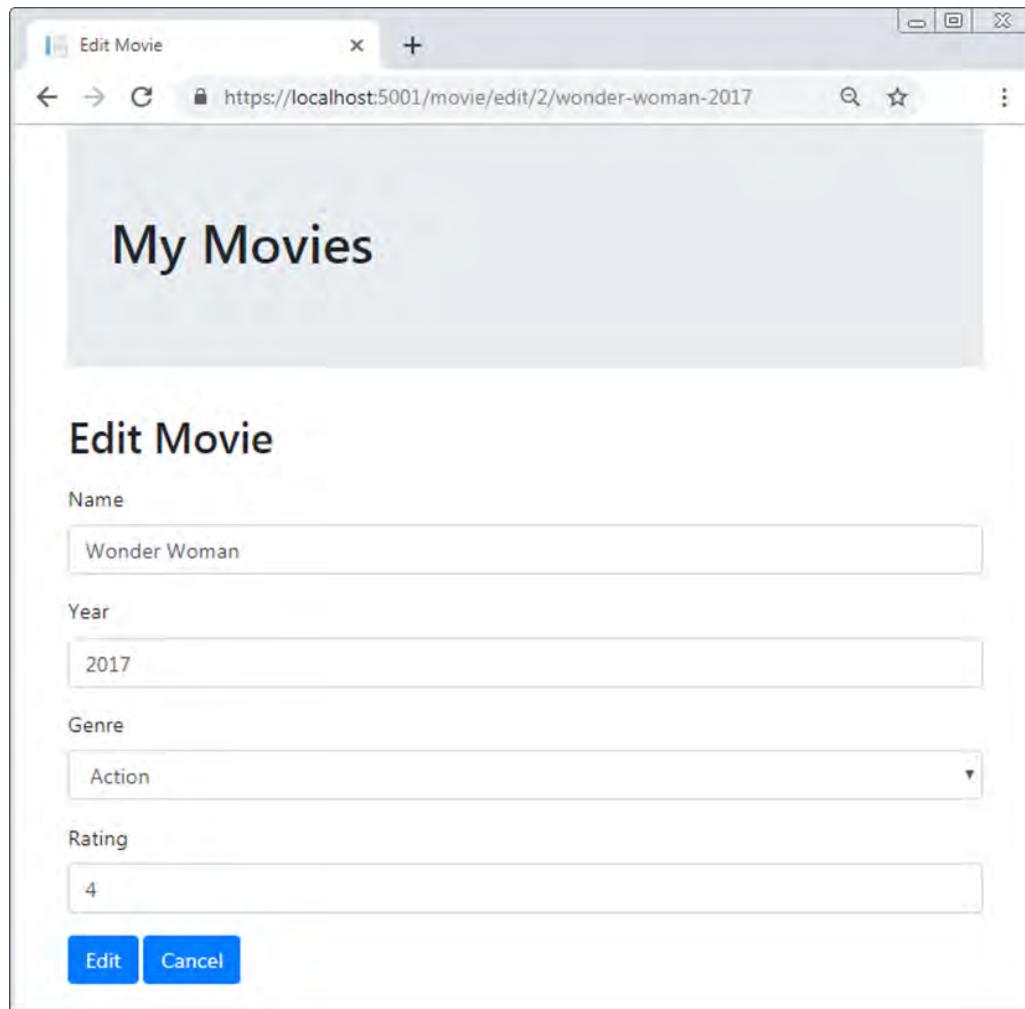
Year:

Genre:

Rating:

Add **Cancel**

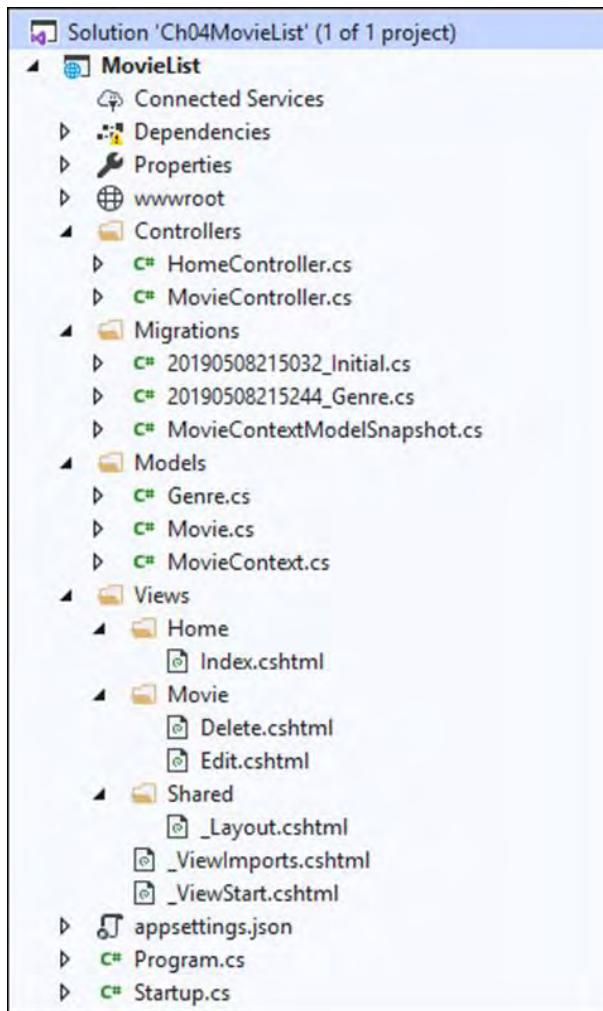
The Edit Movie page of the Movie List app



The Confirm Deletion page of the Movie List app



The Solution Explorer for the Movie List app



Folders in the Movie List app

- Models
- Views
- Controllers
- Migrations

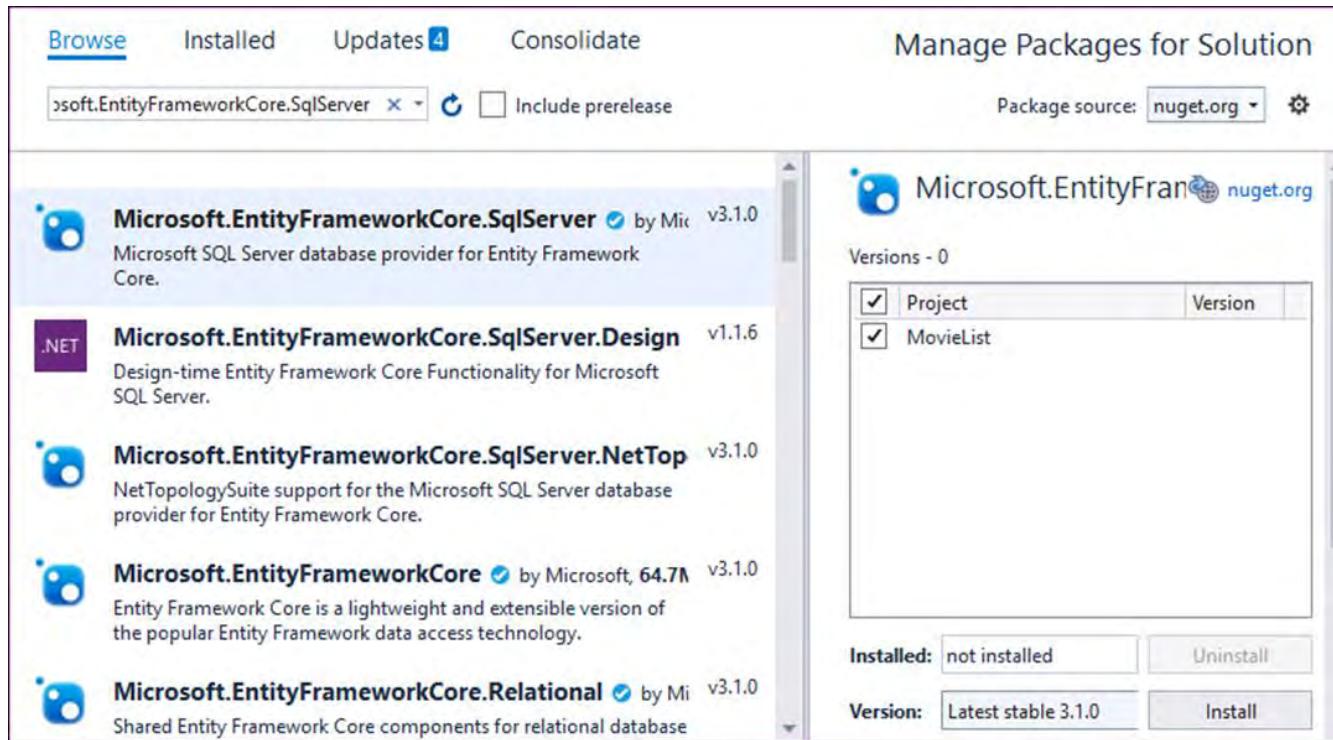
Files in the Movie List app

- appsettings.json
- Startup.cs

How to open the NuGet Package Manager

- Select Tools → Nuget Package Manager → Manage NuGet Packages for Solution.

The NuGet Package Manager



How to install the NuGet packages for EF Core

1. Click the Browse link in the upper left of the window.
2. Type “Microsoft.EntityFrameworkCore.SqlServer” in the search box.
3. Click on the appropriate package from the list that appears in the left-hand panel.
4. In the right-hand panel, check the project name, select the version that matches the version of .NET Core you’re running, and click Install.
5. Review the Preview Changes dialog that comes up and click OK.
6. Review the License Acceptance dialog that comes up and click I Accept.
7. Type “Microsoft.EntityFrameworkCore.Tools” in the search box.
8. Repeat steps 3 through 6.

Three classes provided by EF Core

`DbContext`

`DbContextOptions`

`DbSet<Entity>`

A MovieContext class that inherits the DbContext class

```
using Microsoft.EntityFrameworkCore;

namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        { }

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

A Movie class with a property whose value is generated by the database

```
using System.ComponentModel.DataAnnotations;

namespace MovieList.Models
{
    public class Movie
    {
        // EF Core will configure the database to generate this value
        public int MovieId { get; set; }

        [Required(ErrorMessage = "Please enter a name.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a year.")]
        [Range(1889, 2999, ErrorMessage = "Year must be after 1889.")]
        public int? Year { get; set; }

        [Required(ErrorMessage = "Please enter a rating.")]
        [Range(1, 5, ErrorMessage =
            "Rating must be between 1 and 5.")]
        public int? Rating { get; set; }
    }
}
```

One method of the DbContext class

OnModelCreating(mb)

A MovieContext class that seeds initial Movie data (part 1)

```
namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        { }

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

        protected override void OnModelCreating(
            ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Movie>().HasData(
                new Movie {
                    MovieId = 1,
                    Name = "Casablanca",
                    Year = 1942,
                    Rating = 5
                },

```

A MovieContext class that seeds initial Movie data (part 2)

```
        new Movie {
            MovieId = 2,
            Name = "Wonder Woman",
            Year = 2017,
            Rating = 3
        },
        new Movie {
            MovieId = 3,
            Name = "Moonstruck",
            Year = 1988,
            Rating = 4
        }
    );
}
}
```

A connection string in the appsettings.json file

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "AllowedHosts": "*"  
    "ConnectionStrings": {  
        "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movies;  
                        Trusted_Connection=True;MultipleActiveResultSets=true"  
    }  
}
```

Code that enables dependency injection for DbContext objects

```
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using MovieList.Models;

...
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddDbContext<MovieContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("MovieContext")));
    }
    ...
}
```

The Package Manager Console (PMC) window



How to open the PMC window

- Select the Tools→NuGet Package Manager→Package Manager Console command.

How to create the Movies database from your code

1. Make sure the connection string and dependency injection are set up.
2. Type “Add-Migration Initial” in the PMC at the command prompt and press Enter.
3. Type “Update-Database” at the command prompt and press Enter.

The Up() method of the Initial migration file

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Movies",
        columns: table => new {
            MovieId = table.Column<int>(nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn),
            Name = table.Column<string>(nullable: false),
            Year = table.Column<int>(nullable: false),
            Rating = table.Column<int>(nullable: false)
        },
        constraints: table => {
            table.PrimaryKey("PK_Movies", x => x.MovieId);
        });
}

migrationBuilder.InsertData(
    table: "Movies",
    columns: new[] { "MovieId", "Name", "Rating", "Year" },
    values: new object[] { 1, "Casablanca", 5, 1942 });

    // code that inserts the other two movies
}
```

How to view the database once it's created

1. Choose the View→SQL Server Object Explorer command in Visual Studio.
2. Expand the (localdb)\MSSQLLocalDB node, then expand the Databases node.
3. Expand the Movies node, then expand the Tables node.
4. To view the table columns, expand a table node and then its Columns node.
5. To view the table data, right-click a table and select the ViewData command.

LINQ methods that build or execute a query expression

```
Where(lambda)
OrderBy(lambda)
FirstOrDefault(lambda)
ToList()
```

A method of the DbSet<Entity> class that gets an entity by its id

```
Find(id)
```

A using directive that imports the LINQ namespace

```
using System.Linq;
```

A DbContext property that's used in the examples

```
private MovieContext context { get; set; }
```

Code that builds a query expression

```
IQueryable<Movie> query =  
    context.Movies.OrderBy(m => m.Name);
```

Code that executes a query expression

```
List<Movie> movies = query.ToList();
```

Code that builds and executes a query expression

```
var movies =  
    context.Movies.OrderBy(m => m.Name).ToList();
```

Code that builds a query expression by chaining LINQ methods

```
var query = context.Movies.Where(m => m.Rating > 3)  
    .OrderBy(m => m.Name);
```

Code that builds a query expression on multiple lines

```
IQueryable<Movie> query = context.Movies;  
query = query.Where(m => m.Year > 1970);  
query = query.Where(m => m.Rating > 3);  
query = query.OrderBy(m => m.Name);
```

Three ways to select a movie by its id

```
int id = 1;  
var movie = context.Movies.Where(  
    m => m.MovieId == id).FirstOrDefault();  
var movie = context.Movies.FirstOrDefault(m => m.MovieId == id);  
var movie = context.Movies.Find(id);
```

Three methods of the DbSet class

```
Add(entity)  
Update(entity)  
Remove(entity)
```

One method of the DbContext class

```
SaveChanges()
```

A using directive for the EF Core namespace

```
using Microsoft.EntityFrameworkCore;
```

Code that adds a new movie to the database

```
var movie = new Movie { Name = "Taxi Driver",
                      Year = 1976,
                      Rating = 4
                    };
context.Movies.Add(movie);
context.SaveChanges();
```

An appsettings.json file that displays the generated SQL statements

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information",  
      "Microsoft.EntityFrameworkCore.Database.Command": "Debug"  
    }  
  }  
  ...  
}
```

Code that selects movies from the database

```
var movies = context.Movies.OrderBy(m => m.Name).ToList();
```

The generated SQL statement

```
SELECT [m].[MovieId], [m].[Name], [m].[Rating], [m].[Year]
FROM [Movies] AS [m]
ORDER BY [m].[Name]
```

The Home controller

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using MovieList.Models;

namespace MovieList.Controllers
{
    public class HomeController : Controller
    {
        private MovieContext context { get; set; }

        public HomeController(MovieContext ctx) {
            context = ctx;
        }

        public IActionResult Index() {
            var movies = context.Movies.OrderBy(
                m => m.Name).ToList();
            return View(movies);
        }
    }
}
```

The Home/Index view (part 1)

```
@model List<Movie>
@{
    ViewBag.Title = "My Movies";
}

<h2>Movie List</h2>

<a asp-controller="Movie" asp-action="Add">Add New Movie</a>
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Year</th>
            <th>Rating</th>
            <th></th>
        </tr>
    </thead>
```

The Home/Index view (part 2)

```
<tbody>
    @foreach (var movie in Model) {
        <tr>
            <td>@movie.Name</td>
            <td>@movie.Year</td>
            <td>@movie.Rating</td>
            <td>
                <a asp-controller="Movie" asp-action="Edit"
                    asp-route-id="@movie.MovieId">Edit</a>
                <a asp-controller="Movie" asp-action="Delete"
                    asp-route-id="@movie.MovieId">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>
```

The Movie controller (part 1)

```
namespace MovieList.Controllers
{
    public class MovieController : Controller
    {
        private MovieContext context { get; set; }

        public MovieController(MovieContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IActionResult Add() {
            ViewBag.Action = "Add";
            return View("Edit", new Movie());
        }

        [HttpGet]
        public IActionResult Edit(int id) {
            ViewBag.Action = "Edit";
            var movie = context.Movies.Find(id);
            return View(movie);
        }
    }
}
```

The Movie controller (part 2)

```
[HttpPost]
public IActionResult Edit(Movie movie) {
    if (ModelState.IsValid) {
        if (movie.MovieId == 0)
            context.Movies.Add(movie);
        else
            context.Movies.Update(movie);
        context.SaveChanges();
        return RedirectToAction("Index", "Home");
    } else {
        ViewBag.Action =
            (movie.MovieId == 0) ? "Add": "Edit";
        return View(movie);
    }
}

[HttpGet]
public IActionResult Delete(int id) {
    var movie = context.Movies.Find(id);
    return View(movie);
}
```

The Movie controller (part 3)

```
[HttpPost]
public IActionResult Delete(Movie movie) {
    context.Movies.Remove(movie);
    context.SaveChanges();
    return RedirectToAction("Index", "Home");
}
}
```

The Movie/Edit view (part 1)

```
@model Movie
 @{
    string title = ViewBag.Action + " Movie";
    ViewBag.Title = title;
}

<h2>@ViewBag.Title</h2>

<form asp-action="Edit" method="post">
    <div asp-validation-summary="All"
        class="text-danger">
    </div>

    <div class="form-group">
        <label asp-for="Name">Name</label>
        <input asp-for="Name" class="form-control">
    </div>
```

The Movie/Edit view (part 2)

```
<div class="form-group">
    <label asp-for="Year">Year</label>
    <input asp-for="Year" class="form-control">
</div>

<div class="form-group">
    <label asp-for="Rating">Rating</label>
    <input asp-for="Rating" class="form-control">
</div>

<input type="hidden" asp-for="MovieId" />

<button type="submit" class="btn btn-primary">
    @ViewBag.Action</button>
<a asp-controller="Home" asp-action="Index"
    class="btn btn-primary">Cancel</a>
</form>
```

The Movie/Delete view

```
@model Movie
@{
    ViewBag.Title = "Delete Movie";
}

<h2>Confirm Deletion</h2>
<h3>@Model.Name (@Model.Year)</h3>

<form asp-action="Delete" method="post">
    <input type="hidden" asp-for="MovieId" />

    <button type="submit" class="btn btn-primary">
        Delete</button>
    <a asp-controller="Home" asp-action="Index"
        class="btn btn-primary">Cancel</a>
</form>
```

The Genre class

```
public class Genre
{
    public string GenreId { get; set; }
    public string Name { get; set; }
}
```

How to add a Genre property to the Movie class

```
public class Movie
{
    /* MovieId, Name, Year, and Rating properties
       same as before */

    [Required(ErrorMessage = "Please enter a genre.")]
    public Genre Genre { get; set; }
}
```

How to specify a foreign key property when adding a Genre property

```
public class Movie
{
    /* MovieId, Name, Year, and Rating properties
       same as before */

    [Required(ErrorMessage = "Please enter a genre.")]
    public string GenreId { get; set; }
    public Genre Genre { get; set; }
}
```

A MovieContext class that adds the Genre model with initial data (part 1)

```
public class MovieContext : DbContext {
    public MovieContext(DbContextOptions<MovieContext> options)
        : base(options)
    { }

    public DbSet<Movie> Movies { get; set; }
    public DbSet<Genre> Genres { get; set; }

    protected override void OnModelCreating(
        ModelBuilder modelBuilder) {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Genre>().HasData(
            new Genre { GenreId = "A", Name = "Action" },
            new Genre { GenreId = "C", Name = "Comedy" },
            new Genre { GenreId = "D", Name = "Drama" },
            new Genre { GenreId = "H", Name = "Horror" },
            new Genre { GenreId = "M", Name = "Musical" },
            new Genre { GenreId = "R", Name = "RomCom" },
            new Genre { GenreId = "S", Name = "SciFi" }
        );
    }
}
```

A MovieContext class that adds the Genre model with initial data (part 2)

```
    modelBuilder.Entity<Movie>().HasData(
        new Movie { MovieId = 1, Name = "Casablanca",
                    Year = 1942, Rating = 5, GenreId = "D"
                },
        new Movie { MovieId = 2, Name = "Wonder Woman",
                    Year = 2017, Rating = 3, GenreId = "A"
                },
        new Movie { MovieId = 3, Name = "Moonstruck",
                    Year = 1988, Rating = 4, GenreId = "R"
                }
            );
    }
}
```

How to update the database with the new Genre model and seed data

1. Select Tools → NuGet Package Manager → Package Manager Console to open the Package Manager Console window.
2. Type “Add-Migration Genre” at the command prompt and press Enter.
3. Type “Update-Database” at the command prompt and press Enter.

Some of the code in the Up() method of the Genre migration file (part 1)

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "GenreId",
        table: "Movies",
        nullable: false,
        defaultValue: "");

    migrationBuilder.CreateTable(
        name: "Genres",
        columns: table => new {
            GenreId = table.Column<string>(nullable: false),
            Name = table.Column<string>(nullable: true)
        }, constraints: table => {
            table.PrimaryKey("PK_Genres", x => x.GenreId);
    });
}
```

Some of the code in the Up() method (part 2)

```
migrationBuilder.InsertData(  
    table: "Genres",  
    columns: new[] { "GenreId", "Name" },  
    values: new object[,] {  
        { "A", "Action" },  
        { "C", "Comedy" },  
        { "D", "Drama" },  
        { "H", "Horror" },  
        { "M", "Musical" },  
        { "R", "RomCom" },  
        { "S", "SciFi" }  
  
migrationBuilder.UpdateData(  
    table: "Movies",  
    keyColumn: "MovieId",  
    keyValue: 1,  
    column: "GenreId",  
    value: "D");  
  
// code that updates the other two movies
```

Some of the code in the Up() method (part 3)

```
migrationBuilder.AddForeignKey(  
    name: "FK_Movies_Genres_GenreId",  
    table: "Movies",  
    column: "GenreId",  
    principalTable: "Genres",  
    principalColumn: "GenreId",  
    onDelete: ReferentialAction.Cascade);  
...  
}
```

Another LINQ method that builds
a query expression

```
Include(lambda)
```

The Index() action method of the Home controller

```
using Microsoft.EntityFrameworkCore;
...
public class HomeController : Controller {
...
    public IActionResult Index() {
        var movies = context.Movies.Include(m => m.Genre)
            .OrderBy(m => m.Name).ToList();
        return View(movies);
    }
}
```

The <table> element of the Home/Index view

```
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Year</th>
            <th>Genre</th>
            <th>Rating</th><th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var movie in Model) {
            <tr>
                <td>@movie.Name</td>
                <td>@movie.Year</td>
                <td>@movie.Genre.Name</td>
                <td>@movie.Rating</td>
                <td><!-- Edit/Delete links same as before --></td>
            </tr>
        }
    </tbody>
</table>
```

The Add() action method of the Movie controller

```
[HttpGet]
public IActionResult Add()
{
    ViewBag.Action = "Add";
    ViewBag.Genres
        = context.Genres.OrderBy(g => g.Name).ToList();
    return View("Edit", new Movie());
}
```

The Edit() action method of the Movie controller for GET requests

```
[HttpGet]
public IActionResult Edit(int id) {
    ViewBag.Action = "Edit";
    ViewBag.Genres
        = context.Genres.OrderBy(g => g.Name).ToList();
    var movie = context.Movies.Find(id);
    return View(movie);
}
```

The Edit() action method of the Movie controller for POST requests

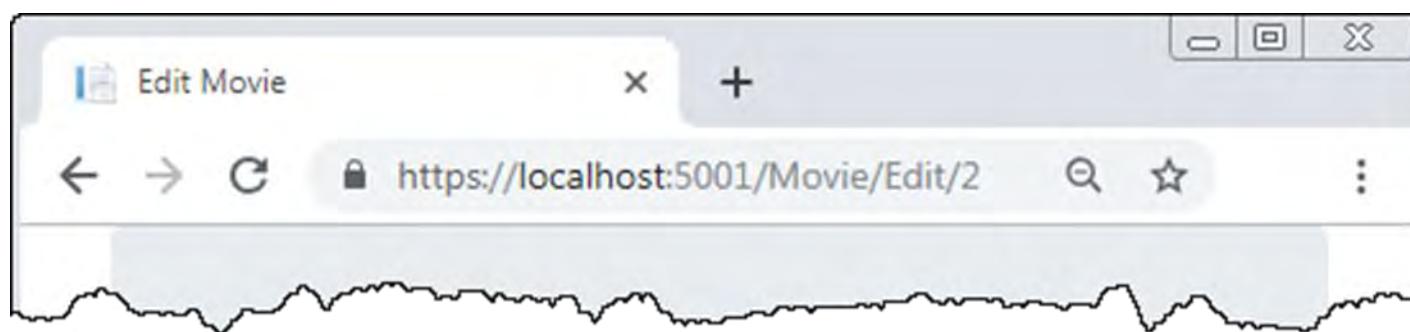
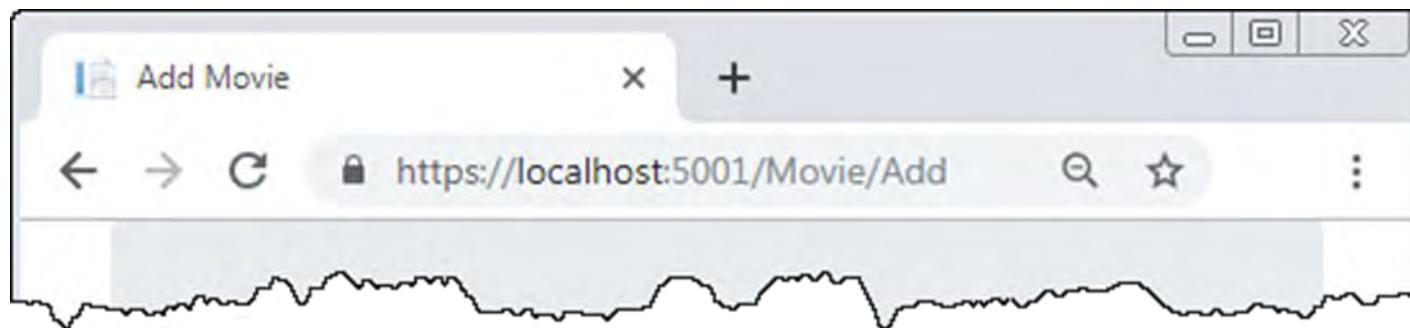
```
[HttpPost]
public IActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        if (movie.MovieId == 0)
            context.Movies.Add(movie);
        else
            context.Movies.Update(movie);
        context.SaveChanges();
        return RedirectToAction("Index", "Home");
    }
    else
    {
        ViewBag.Action = (movie.MovieId == 0) ? "Add": "Edit";
        ViewBag.Genres
            = context.Genres.OrderBy(g => g.Name).ToList();
        return View(movie);
    }
}
```

The form tag of the Movie/Edit view

```
<form asp-action="Edit" method="post">
    ...
    <div class="form-group">
        <label asp-for="GenreId">Genre</label>
        <select asp-for="GenreId" class="form-control">
            <option value="">select a genre</option>
            @foreach (Genre g in ViewBag.Genres)
            {
                <option value="@g.GenreId">
                    @g.Name
                </option>
            }
        </select>
    </div>
    ...

```

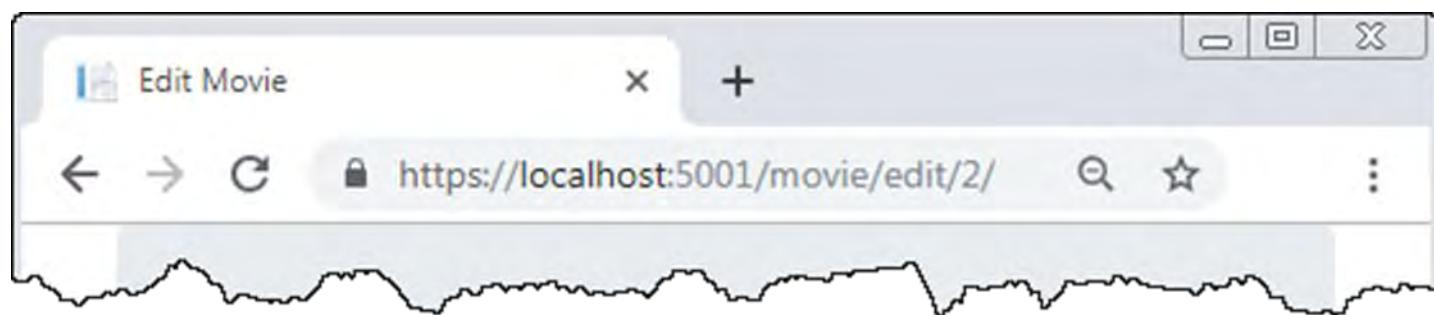
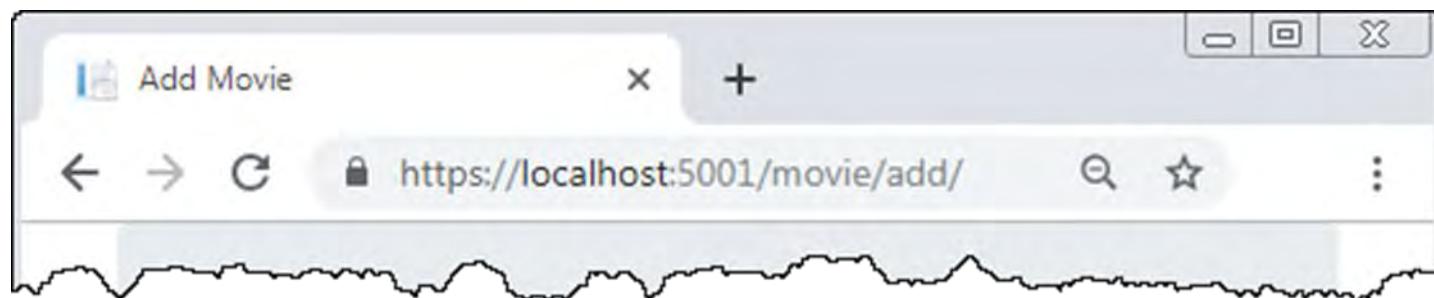
The default URLs of an MVC app



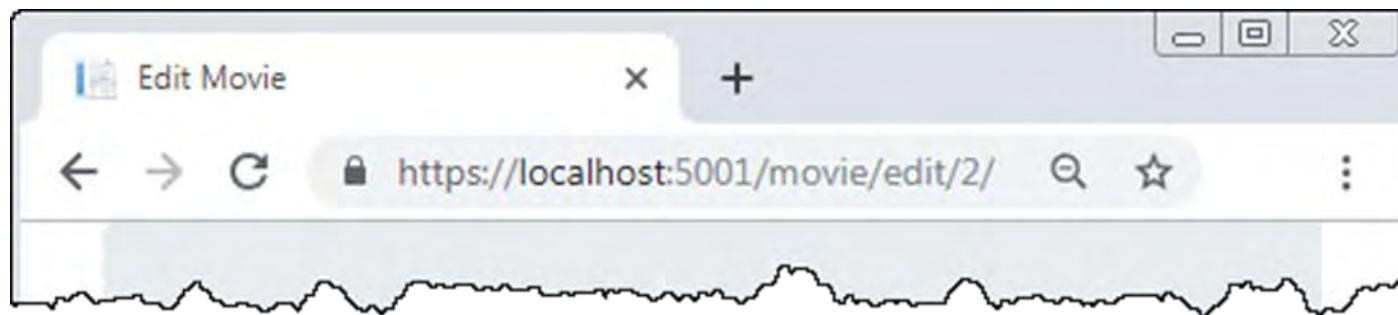
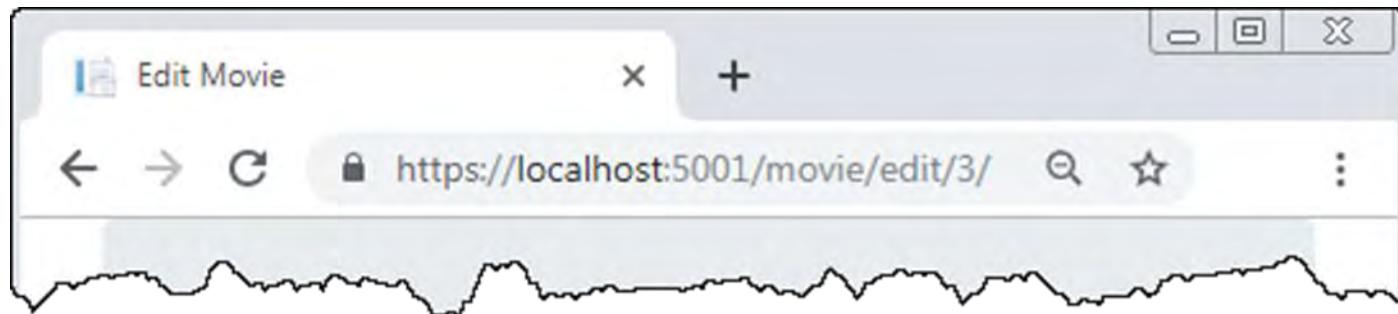
The ConfigureServices() method to make URLs lowercase with a trailing slash

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddRouting(options => {
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
    });
    ...
}
```

The reconfigured URLs



The Edit page with numeric ID values only in the URL



The default route updated to include
a second optional parameter

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

A read-only property named Slug
in the Movie class

```
public string Slug =>
    Name?.Replace(' ', '-').ToLower() + '-' +
    Year?.ToString();
```

The Edit/Delete links in the Home/Index view

```
<a asp-controller="Movie" asp-action="Edit"  
    asp-route-id="@movie.MovieId"  
    asp-route-slug="@movie.Slug">Edit</a>  
<a asp-controller="Movie" asp-action="Delete"  
    asp-route-id="@movie.MovieId"  
    asp-route-slug="@movie.Slug">Delete</a>
```

The Edit page with a slug in the URL



XDocument.Load Method

Reference

[Is this page helpful?](#)

Definition

Namespace: [System.Xml.Linq](#)

Assembly: System.Xml.XDocument.dll

Creates a new [XDocument](#) from a file specified by a URI, from an [TextReader](#), or from an [XmlReader](#).

In this article

[Definition](#)

[Overloads](#)

[Remarks](#)

[Load\(Stream\)](#)

[Load\(TextReader\)](#)

[Load\(String\)](#)

[Load\(XmlReader\)](#)

[Load\(Stream, LoadOptions\)](#)

[Load\(TextReader, LoadOptions\)](#)

[Load\(String, LoadOptions\)](#)

[Load\(XmlReader, LoadOptions\)](#)

Overloads

[Load\(Stream\)](#)

Creates a new [XDocument](#) instance by using the specified stream.

[Load\(TextReader\)](#)

Creates a new [XDocument](#) from a [TextReader](#).

[Load\(String\)](#)

Creates a new [XDocument](#) from a file.

Load(XmlReader)	Creates a new XDocument from an XmlReader .
Load(Stream, LoadOptions)	Creates a new XDocument instance by using the specified stream, optionally preserving white space, setting the base URI, and retaining line information.
Load(TextReader, LoadOptions)	Creates a new XDocument from a TextReader , optionally preserving white space, setting the base URI, and retaining line information.
Load(String, LoadOptions)	Creates a new XDocument from a file, optionally preserving white space, setting the base URI, and retaining line information.
Load(XmlReader, LoadOptions)	Loads an XDocument from an XmlReader , optionally setting the base URI, and retaining line information.

Remarks

Using one of the overloads of this method, you can load an [XDocument](#) from a file, a [TextReader](#), or an [XmlReader](#).

To create an [XDocument](#) from a string that contains XML, use [Parse](#).

Load(Stream)

Creates a new [XDocument](#) instance by using the specified stream.

C#

 Copy

```
public static System.Xml.Linq.XDocument Load (System.IO.Stream stream);
```

Parameters

stream [Stream](#)

The stream that contains the XML data.

Returns

[XDocument](#)

An [XDocument](#) object that reads the data that is contained in the stream.

Remarks

If you want to control load options, use the [Load](#) overload that takes [LoadOptions](#) as a parameter.

The loading functionality of LINQ to XML is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

If you have to modify [XmlReaderSettings](#), follow these steps:

1. Create an [XmlReader](#) by calling one of the [Create](#) overloads that take [XmlReaderSettings](#) as a parameter.
2. Pass the [XmlReader](#) to one of the [Load](#) overloads of [XDocument](#) that takes [XmlReader](#) as a parameter.

Applies to

- .NET 6 and other versions

Load(TextReader)

Creates a new [XDocument](#) from a [TextReader](#).

C#

 Copy

```
public static System.Xml.Linq.XDocument Load (System.IO.TextReader  
textReader);
```

Parameters

textReader [TextReader](#)

A [TextReader](#) that contains the content for the [XDocument](#).

Returns

[XDocument](#)

An [XDocument](#) that contains the contents of the specified [TextReader](#).

Examples

The following example creates a document from a [StringReader](#).

C#

 Copy

```
TextReader tr = new StringReader("<Root>Content</Root>");  
XDocument doc = XDocument.Load(tr);  
Console.WriteLine(doc);
```

This example produces the following output:

XML

 Copy

```
<Root>Content</Root>
```

Remarks

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [Parse](#)
- [Save](#)
- [WriteTo\(XmlWriter\)](#)
- [LINQ to XML overview](#)
- [Query an XDocument vs. query an XElement](#)

Applies to

- .NET 6 and other versions

Load(String)

Creates a new [XDocument](#) from a file.

C#	 Copy
----	--

```
public static System.Xml.Linq.XDocument Load (string uri);
```

Parameters

uri [String](#)

A URI string that references the file to load into a new [XDocument](#).

Returns

[XDocument](#)

An [XDocument](#) that contains the contents of the specified file.

Examples

The following example shows how to load an [XDocument](#) from a file.

This example uses the following XML document:

[Sample XML File: Typical Purchase Order \(LINQ to XML\)](#)

C#	 Copy
----	--

```
XDocument doc = XDocument.Load("PurchaseOrder.xml");
Console.WriteLine(doc);
```

This example produces the following output:

 Copy

```
<PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
  <Address Type="Shipping">
    <Name>Ellen Adams</Name>
    <Street>123 Maple Street</Street>
    <City>Mill Valley</City>
    <State>CA</State>
    <Zip>10999</Zip>
    <Country>USA</Country>
  </Address>
  <Address Type="Billing">
    <Name>Tai Yee</Name>
    <Street>8 Oak Avenue</Street>
    <City>Old Town</City>
    <State>PA</State>
    <Zip>95819</Zip>
    <Country>USA</Country>
  </Address>
  <DeliveryNotes>Please leave packages in shed by driveway.
  </DeliveryNotes>
  <Items>
    <Item PartNumber="872-AA">
      <ProductName>Lawnmower</ProductName>
      <Quantity>1</Quantity>
      <USPrice>148.95</USPrice>
      <Comment>Confirm this is electric</Comment>
    </Item>
    <Item PartNumber="926-AA">
      <ProductName>Baby Monitor</ProductName>
      <Quantity>2</Quantity>
      <USPrice>39.98</USPrice>
      <ShipDate>1999-05-21</ShipDate>
    </Item>
  </Items>
</PurchaseOrder>
```

Remarks

This method uses an underlying [XmlReader](#) to read the XML into an XML tree.

Use [Parse](#) to create an [XDocument](#) from a string that contains XML.

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [Parse](#)
- [Save](#)
- [WriteTo\(XmlWriter\)](#)
- [LINQ to XML overview](#)
- [Query an XDocument vs. query an XElement](#)

Applies to

- ▶ .NET 6 and other versions

Load(XmlReader)

Creates a new [XDocument](#) from an [XmlReader](#).

C#

 Copy

```
public static System.Xml.Linq.XDocument Load (System.Xml.XmlReader  
reader);
```

Parameters

reader [XmlReader](#)

A [XmlReader](#) that contains the content for the [XDocument](#).

Returns

[XDocument](#)

An [XDocument](#) that contains the contents of the specified [XmlReader](#).

Examples

The following example creates a DOM document, creates an [XmlNodeReader](#) from the DOM document, creates an [XDocument](#) using the [XmlNodeReader](#).

C#	 Copy
----	--

```
// Create a DOM document with some content.  
XmlDocument doc = new XmlDocument();  
XmlElement child = doc.CreateElement("Child");  
child.InnerText = "child contents";  
XmlElement root = doc.CreateElement("Root");  
root.AppendChild(child);  
doc.AppendChild(root);  
  
// create a reader and move to the content  
using (XmlNodeReader nodeReader = new XmlNodeReader(doc)) {  
    // the reader must be in the Interactive state in order to  
    // create a LINQ to XML tree from it.  
    nodeReader.MoveToContent();  
  
    XDocument xRoot = XDocument.Load(nodeReader);  
    Console.WriteLine(xRoot);  
}
```

This example produces the following output:

XML	 Copy
-----	---

```
<Root>  
  <Child>child contents</Child>  
</Root>
```

Remarks

One possible use for this method is to create a copy of a DOM document in a LINQ to XML tree. To do this, you create an [XmlNodeReader](#) from a DOM document, and then use the [XmlNodeReader](#) to create an [XDocument](#).

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [Parse](#)
- [Save](#)

- [WriteTo\(XmlWriter\)](#)
- [LINQ to XML overview](#)
- [Query an XDocument vs. query an XElement](#)

Applies to

- ▶ .NET 6 and other versions

Load(Stream, LoadOptions)

Creates a new [XDocument](#) instance by using the specified stream, optionally preserving white space, setting the base URI, and retaining line information.

C#

 Copy

```
public static System.Xml.Linq.XDocument Load (System.IO.Stream stream,  
System.Xml.Linq.LoadOptions options);
```

Parameters

stream [Stream](#)

The stream containing the XML data.

options [LoadOptions](#)

A [LoadOptions](#) that specifies whether to load base URI and line information.

Returns

[XDocument](#)

An [XDocument](#) object that reads the data that is contained in the stream.

Remarks

The loading functionality of LINQ to XML is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload

methods and the [XmlReader](#) methods that read and parse the document.

If you have to modify [XmlReaderSettings](#), follow these steps:

1. Create an [XmlReader](#) by calling one of the [Create](#) overloads that takes [XmlReaderSettings](#) as a parameter.
2. Pass the [XmlReader](#) to one of the [Load](#) overloads of [XDocument](#) that takes [XmlReader](#) as a parameter.

Applies to

- ▶ .NET 6 and other versions

Load(TextReader, LoadOptions)

Creates a new [XDocument](#) from a [TextReader](#), optionally preserving white space, setting the base URI, and retaining line information.

C#	 Copy
----	--

```
public static System.Xml.Linq.XDocument Load (System.IO.TextReader  
textReader, System.Xml.Linq.LoadOptions options);
```

Parameters

textReader [TextReader](#)

A [TextReader](#) that contains the content for the [XDocument](#).

options [LoadOptions](#)

A [LoadOptions](#) that specifies white space behavior, and whether to load base URI and line information.

Returns

[XDocument](#)

An [XDocument](#) that contains the XML that was read from the specified [TextReader](#).

Examples

The following example creates a document from a [StringReader](#).

C#	 Copy
<pre>TextReader sr; int whiteSpaceNodes; sr = new StringReader("<Root> <Child> </Child> </Root>"); XDocument xmlTree1 = XDocument.Load(sr, LoadOptions.None); sr.Close(); whiteSpaceNodes = xmlTree1 .Element("Root") .DescendantNodesAndSelf() .OfType<XText>() .Where(tNode => tNode.ToString().Trim().Length == 0) .Count(); Console.WriteLine("Count of white space nodes (not preserving whitespace): {0}", whiteSpaceNodes); sr = new StringReader("<Root> <Child> </Child> </Root>"); XDocument xmlTree2 = XDocument.Load(sr, LoadOptions.PreserveWhitespace); sr.Close(); whiteSpaceNodes = xmlTree2 .Element("Root") .DescendantNodesAndSelf() .OfType<XText>() .Where(tNode => tNode.ToString().Trim().Length == 0) .Count(); Console.WriteLine("Count of white space nodes (preserving whitespace): {0}", whiteSpaceNodes);</pre>	

This example produces the following output:

 Copy
<pre>Count of white space nodes (not preserving whitespace): 0 Count of white space nodes (preserving whitespace): 3</pre>

Remarks

If the source XML is indented, setting the [PreserveWhitespace](#) flag in `options` causes the reader to read all white space in the source XML. Nodes of type [XText](#) are created for both significant and insignificant white space.

If the source XML is indented, not setting the [PreserveWhitespace](#) flag in `options` causes the reader to ignore all of the insignificant white space in the source XML. The XML tree is created without any text nodes for insignificant white space.

If the source XML is not indented, setting the [PreserveWhitespace](#) flag in `options` has no effect. Significant white space is still preserved, and there are no spans of insignificant white space that could cause the creation of more white space text nodes.

For more information, see [Preserve white space while loading or parsing XML](#) and [Preserve white space while serializing](#).

Use [Parse](#) to create an [XElement](#) from a string that contains XML.

Setting [SetBaseUri](#) is not valid when loading from a [TextReader](#).

There is a performance penalty if you set the [SetLineInfo](#) flag.

The line information is accurate immediately after loading the XML document. If you modify the XML tree after loading the document, the line information may become meaningless.

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [Parse](#)
- [Save](#)
- [WriteTo\(XmlWriter\)](#)
- [LINQ to XML overview](#)
- [Query an XDocument vs. query an XElement](#)

Applies to

- .NET 6 and other versions

Load(String, LoadOptions)

Creates a new [XDocument](#) from a file, optionally preserving white space, setting the base URI, and retaining line information.

C#

 Copy

```
public static System.Xml.Linq.XDocument Load (string uri,  
System.Xml.Linq.LoadOptions options);
```

Parameters

uri [String](#)

A URI string that references the file to load into a new [XDocument](#).

options [LoadOptions](#)

A [LoadOptions](#) that specifies white space behavior, and whether to load base URI and line information.

Returns

[XDocument](#)

An [XDocument](#) that contains the contents of the specified file.

Examples

The following example shows how to load an [XDocument](#) from a file.

This example uses the following XML document:

[Sample XML File: Typical Purchase Order \(LINQ to XML\)](#)

C#

 Copy

```
XDocument doc1 = XDocument.Load("PurchaseOrder.xml", LoadOptions.None);
```

```
Console.WriteLine("nodes if not preserving whitespace: {0}",  
    doc1.DescendantNodes().Count());  
  
XDocument doc2 = XDocument.Load("PurchaseOrder.xml",  
    LoadOptions.PreserveWhitespace);  
Console.WriteLine("nodes if preserving whitespace: {0}",  
    doc2.DescendantNodes().Count());
```

This example produces the following output:

 Copy

```
nodes if not preserving whitespace: 48  
nodes if preserving whitespace: 82
```

Remarks

If the source XML is indented, setting the [PreserveWhitespace](#) flag in `options` causes the reader to read all white space in the source XML. Nodes of type [XText](#) are created for both significant and insignificant white space.

If the source XML is indented, not setting the [PreserveWhitespace](#) flag in `options` causes the reader to ignore all of the insignificant white space in the source XML. The XML tree is created without any text nodes for insignificant white space.

If the source XML is not indented, setting the [PreserveWhitespace](#) flag in `options` has no effect. Significant white space is still preserved, and there are no spans of insignificant white space that could cause the creation of more white space text nodes.

For more information, see [Preserve white space while loading or parsing XML](#) and [Preserve white space while serializing](#).

Use [Parse](#) to create an [XDocument](#) from a string that contains XML.

There is a performance penalty if you set the [SetBaseUri](#) and the [SetLineInfo](#) flags.

The base URI and the line information are accurate immediately after loading the XML document. If you modify the XML tree after loading the document, the base URI and line information may become meaningless.

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [Parse](#)
- [Save](#)
- [WriteTo\(XmlWriter\)](#)
- [LINQ to XML overview](#)
- [Query an XDocument vs. query an XElement](#)

Applies to

- ▶ .NET 6 and other versions

Load(XmlReader, LoadOptions)

Loads an [XDocument](#) from an [XmlReader](#), optionally setting the base URI, and retaining line information.

C#	 Copy
----	--

```
public static System.Xml.Linq.XDocument Load (System.Xml.XmlReader  
reader, System.Xml.Linq.LoadOptions options);
```

Parameters

reader [XmlReader](#)

A [XmlReader](#) that will be read for the content of the [XDocument](#).

options [LoadOptions](#)

A [LoadOptions](#) that specifies whether to load base URI and line information.

Returns

XDocument

An [XDocument](#) that contains the XML that was read from the specified [XmlReader](#).

Examples

The following example loads the line information that it loads from the [XmlReader](#). It then prints the line information.

C#

 Copy

```
string markup =
@"<Root>
  <Child>
    <GrandChild/>
  </Child>
</Root>";

// Create a reader and move to the content.
using (XmlReader nodeReader = XmlReader.Create(new StringReader(markup)))
{
    // the reader must be in the Interactive state in order to
    // Create a LINQ to XML tree from it.
    nodeReader.MoveToContent();

    XDocument xRoot = XDocument.Load(nodeReader,
LoadOptions.SetLineInfo);
    Console.WriteLine("{0}{1}{2}",
        "Element Name".PadRight(20),
        "Line".PadRight(5),
        "Position");
    Console.WriteLine("{0}{1}{2}",
        "-----".PadRight(20),
        "----".PadRight(5),
        "----");
    foreach ( XElement e in xRoot.Elements("Root").DescendantsAndSelf())
        Console.WriteLine("{0}{1}{2}",
            ("").PadRight(e.Ancestors().Count() * 2) +
e.Name).PadRight(20),
            ((IXmlLineInfo)e).LineNumber.ToString().PadRight(5),
            ((IXmlLineInfo)e).LinePosition);
}
```

This example produces the following output:

 Copy

Element Name	Line	Position
Root	1	2
Child	2	6
GrandChild	3	10

Remarks

By creating an [XmlNodeReader](#) from a DOM document, and then using the [XmlNodeReader](#) to create an [XElement](#), this method can be used to create a copy of a DOM document in a LINQ to XML tree.

Use [Parse](#) to create an [XDocument](#) from a string that contains XML.

Setting [PreserveWhitespace](#) is not valid when loading from a [XmlReader](#). The [XmlReader](#) will be configured to either read whitespace or not. The LINQ to XML tree will be populated with the whitespace nodes that the reader surfaces. This will be the behavior regardless of whether [PreserveWhitespace](#) is set or not.

The [XmlReader](#) may have a valid base URI or not. If you set [SetBaseUri](#), the base URI will be set in the XML tree from the base URI that is reported by the [XmlReader](#).

The [XmlReader](#) may have a valid line information or not. If you set [SetLineInfo](#), the line information will be set in the XML tree from the line information that is reported by the [XmlReader](#).

There is a performance penalty if you set the [SetLineInfo](#) flag.

The line information is accurate immediately after loading the XML document. If you modify the XML tree after loading the document, the line information may become meaningless.

LINQ to XML's loading functionality is built upon [XmlReader](#). Therefore, you might catch any exceptions that are thrown by the [XmlReader.Create](#) overload methods and the [XmlReader](#) methods that read and parse the document.

See also

- [WriteTo\(XmlWriter\)](#)

- [Save](#)
- [Parse](#)
- [LINQ to XML overview](#)
- [How to read and write an encoded document](#)

Applies to

- ▶ .NET 6 and other versions

Recommended content

[XElement.Attributes Method \(System.Xml.Linq\)](#)

Returns a collection of attributes of this element.

[XElement.Load Method \(System.Xml.Linq\)](#)

Creates a new XElement from a file specified by a URI, from an TextReader, or from an XmlReader.

[XDocument Constructor \(System.Xml.Linq\)](#)

Initializes a new instance of the XDocument class.

[XAttribute Class \(System.Xml.Linq\)](#)

Represents an XML attribute.

Show more ▾

Get Started with ASP.NET Web API 2 (C#)

Article • 03/10/2020 • 8 minutes to read •  +7

[Is this page helpful?](#)

In this article

[Software versions used in the tutorial](#)

[Create a Web API Project](#)

[Adding a Model](#)

[Adding a Controller](#)

[Calling the Web API with Javascript and jQuery](#)

[Running the Application](#)

[Using F12 to View the HTTP Request and Response](#)

[See this App Running on Azure](#)

[Next Steps](#)

by [Mike Wasson](#)

[Download Completed Project](#)

In this tutorial, you will use ASP.NET Web API to create a web API that returns a list of products.

HTTP is not just for serving up web pages. HTTP is also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

ASP.NET Web API is a framework for building web APIs on top of the .NET Framework.

Software versions used in the tutorial

- [Visual Studio 2017](#)
- [Web API 2](#)

See [Create a web API with ASP.NET Core and Visual Studio for Windows](#) for a newer version of this tutorial.

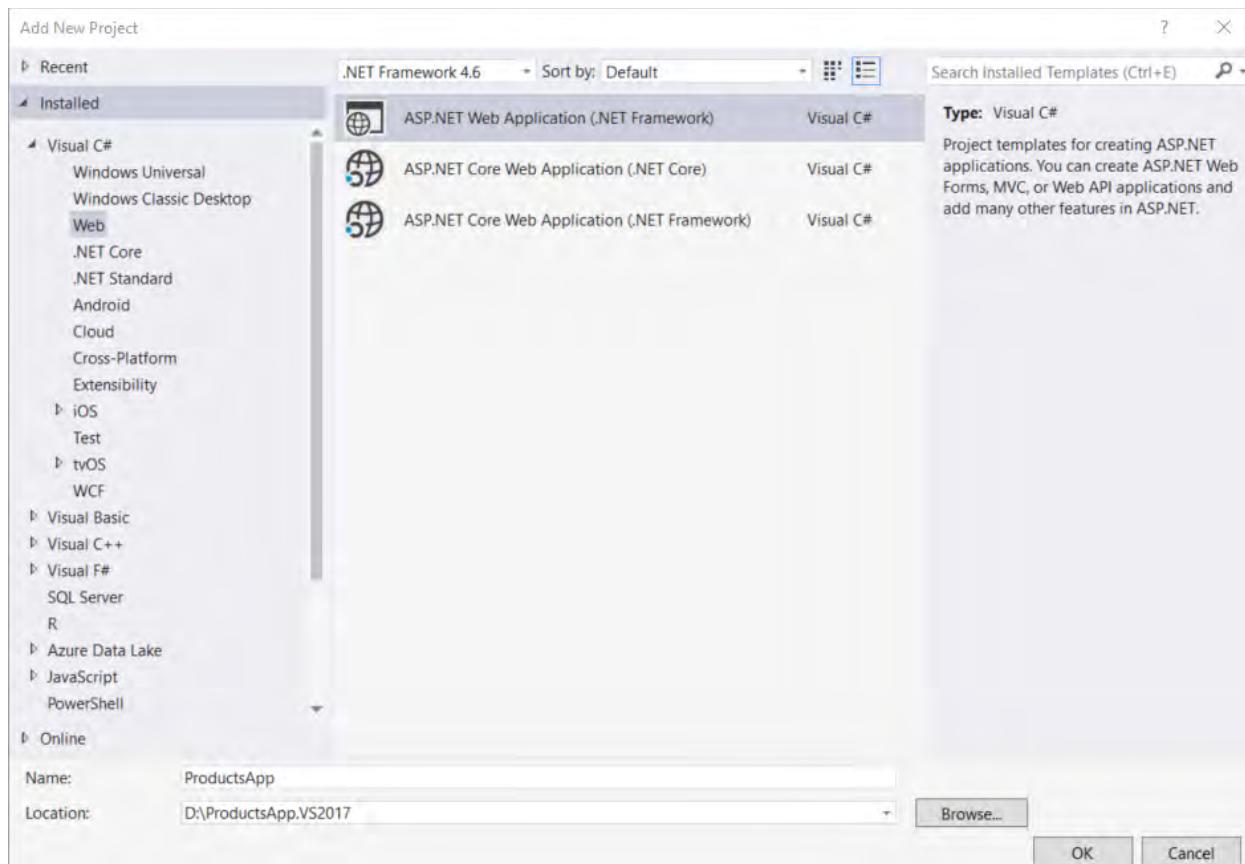
Create a Web API Project

In this tutorial, you will use ASP.NET Web API to create a web API that returns a list of products. The front-end web page uses jQuery to display the results.

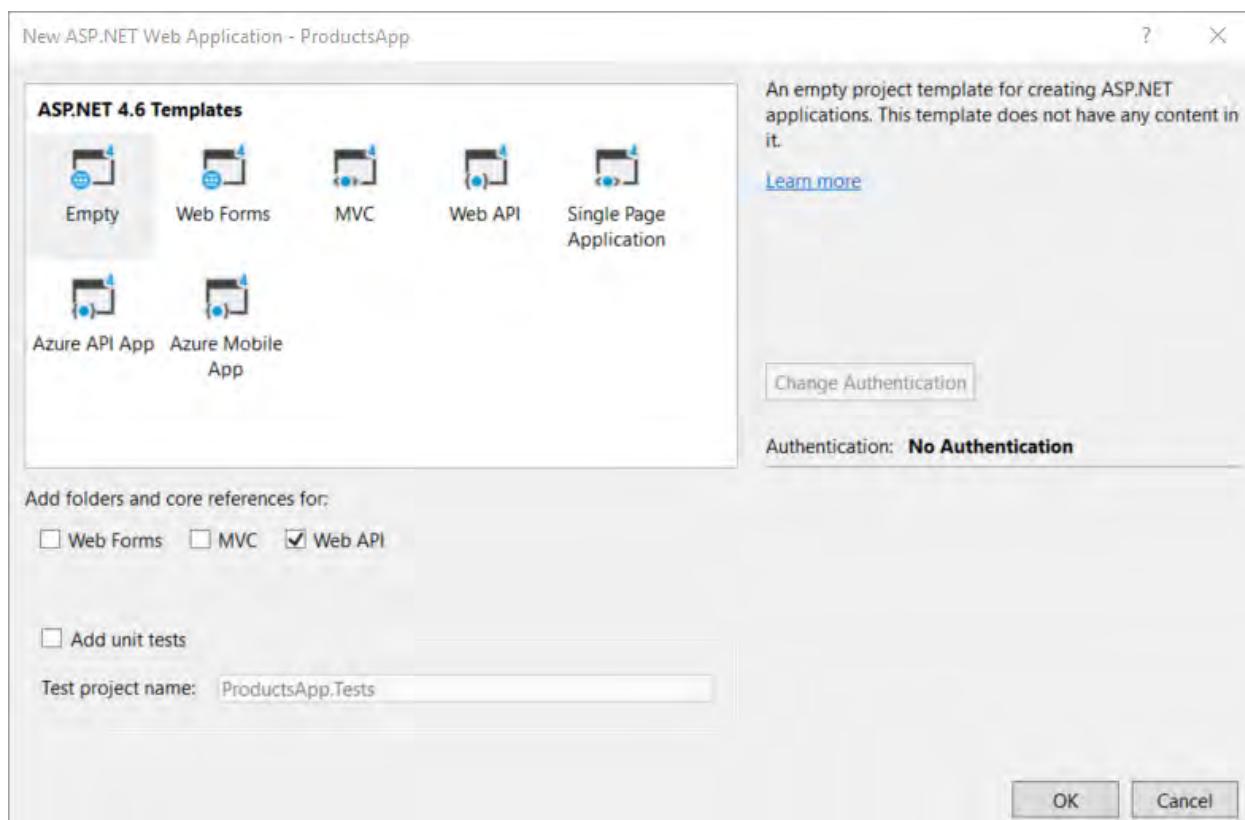


Start Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C# node**. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET Web Application**. Name the project "ProductsApp" and click **OK**.



In the **New ASP.NET Project** dialog, select the **Empty** template. Under "Add folders and core references for", check **Web API**. Click **OK**.



① Note

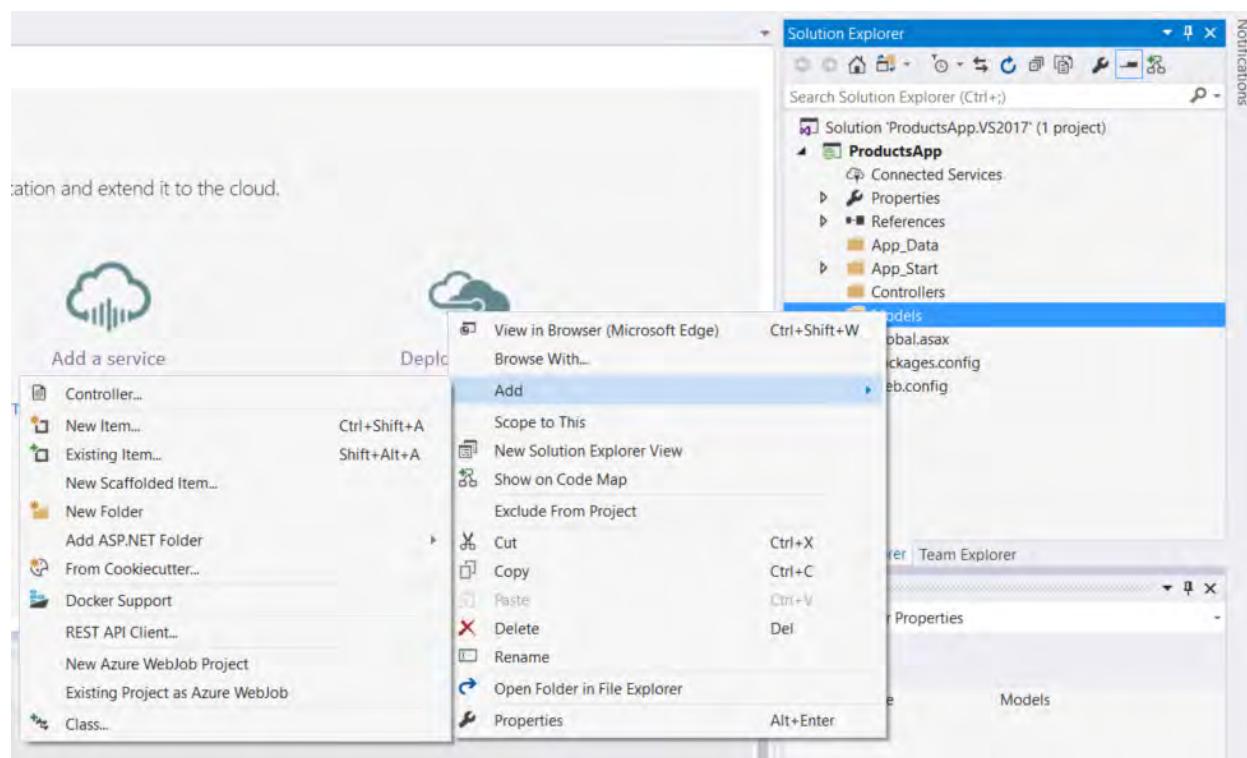
You can also create a Web API project using the "Web API" template. The Web API template uses ASP.NET MVC to provide API help pages. I'm using the Empty template for this tutorial because I want to show Web API without MVC. In general, you don't need to know ASP.NET MVC to use Web API.

Adding a Model

A *model* is an object that represents the data in your application. ASP.NET Web API can automatically serialize your model to JSON, XML, or some other format, and then write the serialized data into the body of the HTTP response message. As long as a client can read the serialization format, it can deserialize the object. Most clients can parse either XML or JSON. Moreover, the client can indicate which format it wants by setting the Accept header in the HTTP request message.

Let's start by creating a simple model that represents a product.

If Solution Explorer is not already visible, click the **View** menu and select **Solution Explorer**. In Solution Explorer, right-click the Models folder. From the context menu, select **Add** then select **Class**.



Name the class "Product". Add the following properties to the `Product` class.

C#

 Copy

```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

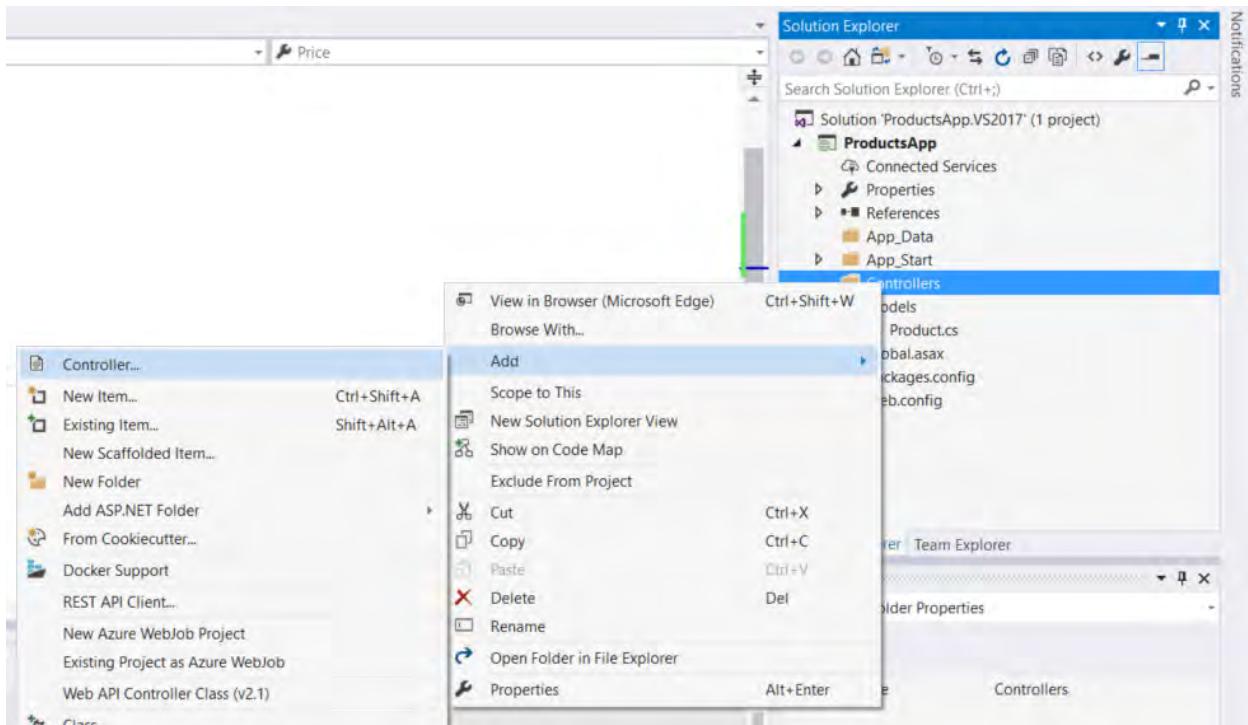
Adding a Controller

In Web API, a *controller* is an object that handles HTTP requests. We'll add a controller that can return either a list of products or a single product specified by ID.

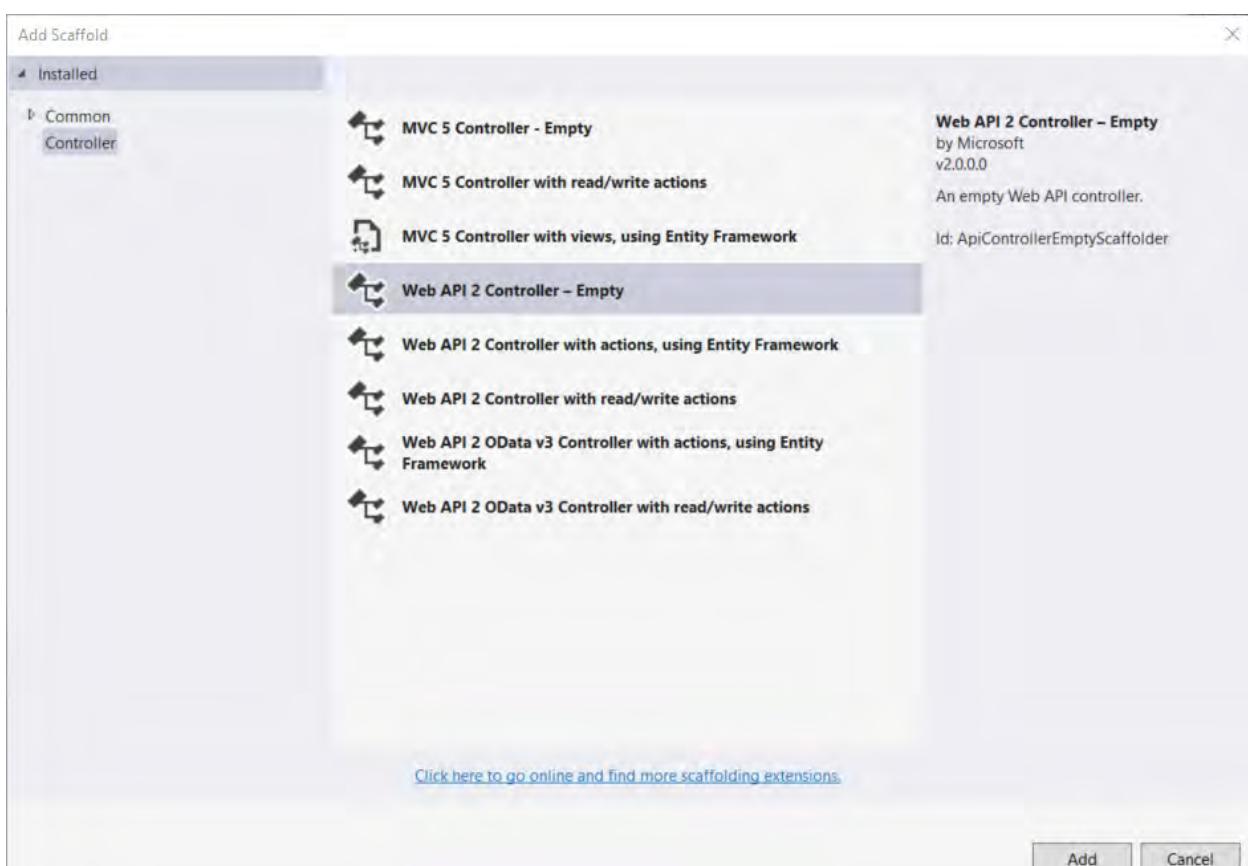
 **Note**

If you have used ASP.NET MVC, you are already familiar with controllers. Web API controllers are similar to MVC controllers, but inherit the `ApiController` class instead of the `Controller` class.

In **Solution Explorer**, right-click the Controllers folder. Select **Add** and then select **Controller**.



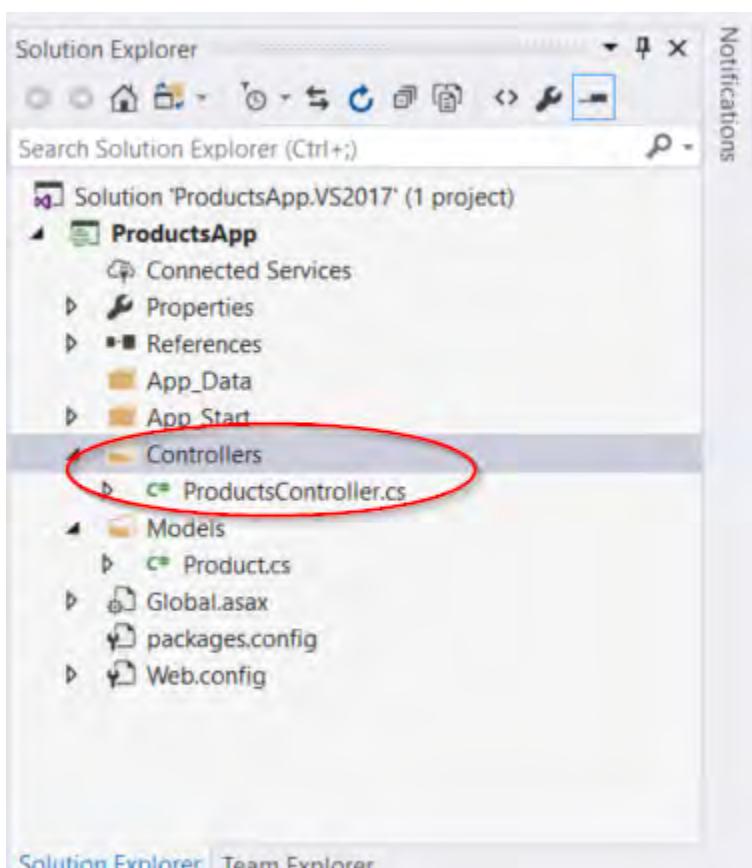
In the **Add Scaffold** dialog, select **Web API Controller - Empty**. Click **Add**.



In the **Add Controller** dialog, name the controller "ProductsController". Click **Add**.



The scaffolding creates a file named `ProductsController.cs` in the `Controllers` folder.



① Note

You don't need to put your controllers into a folder named `Controllers`. The folder name is just a convenient way to organize your source files.

If this file is not open already, double-click the file to open it. Replace the code in this file with the following:

C#

Copy

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category =
"Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price =
3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware",
Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

To keep the example simple, products are stored in a fixed array inside the controller class. Of course, in a real application, you would query a database or use some other external data source.

The controller defines two methods that return products:

- The `GetAllProducts` method returns the entire list of products as an `IEnumerable<Product>` type.
- The `GetProduct` method looks up a single product by its ID.

That's it! You have a working web API. Each method on the controller corresponds to

one or more URLs:

Controller Method	URI
GetAllProducts	/api/products
GetProduct	/api/products/ <i>id</i>

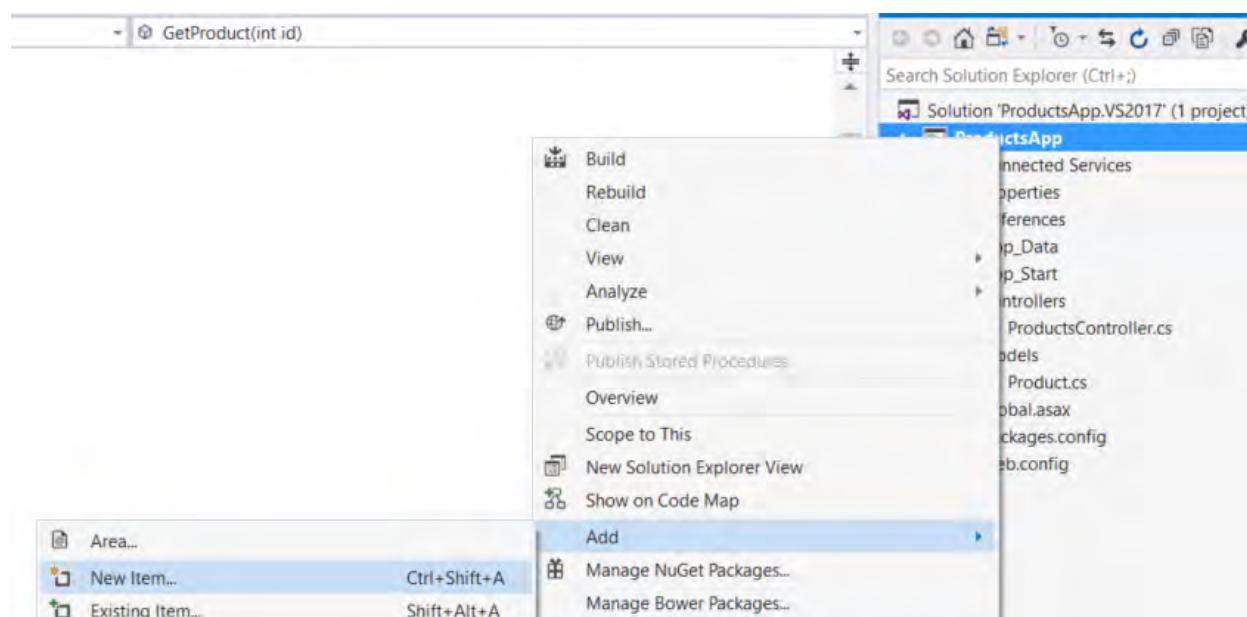
For the `GetProduct` method, the `id` in the URI is a placeholder. For example, to get the product with ID of 5, the URI is `api/products/5`.

For more information about how Web API routes HTTP requests to controller methods, see [Routing in ASP.NET Web API](#).

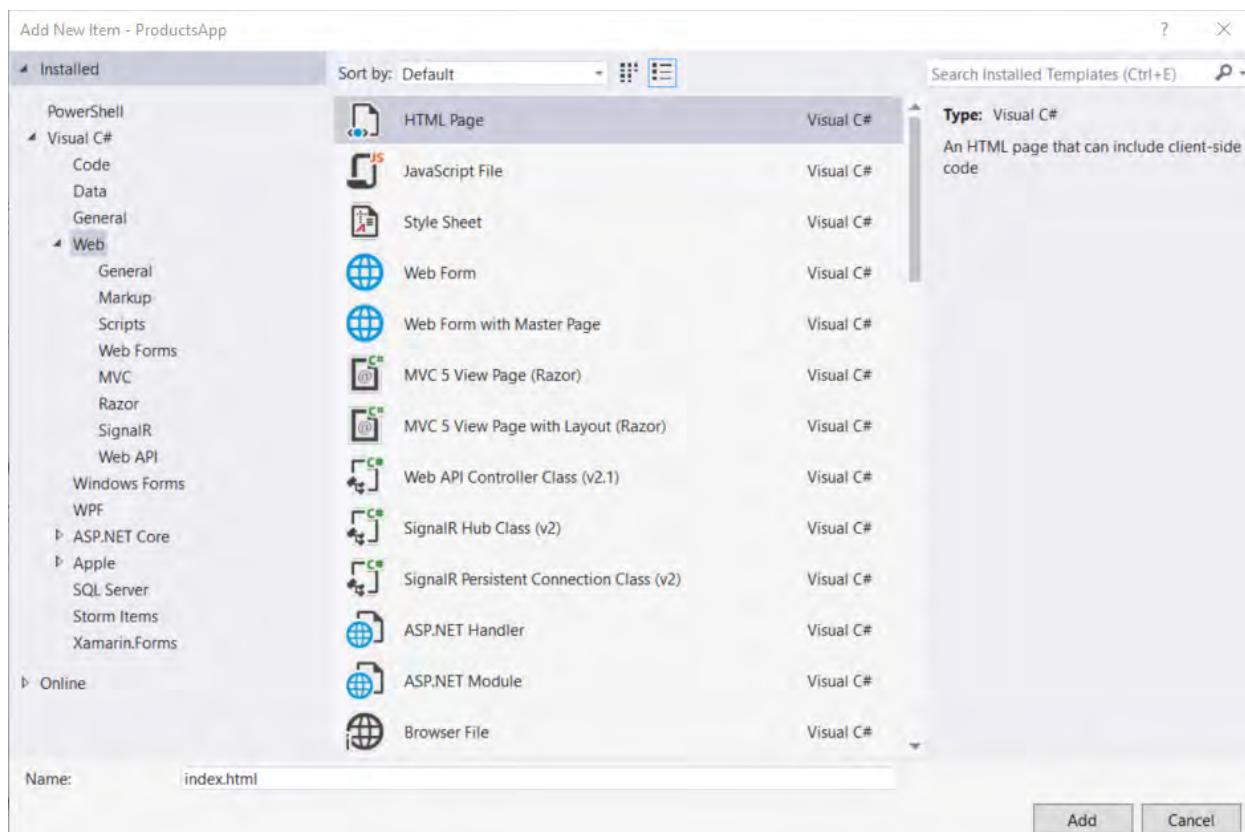
Calling the Web API with Javascript and jQuery

In this section, we'll add an HTML page that uses AJAX to call the web API. We'll use jQuery to make the AJAX calls and also to update the page with the results.

In Solution Explorer, right-click the project and select **Add**, then select **New Item**.



In the **Add New Item** dialog, select the **Web** node under **Visual C#**, and then select the **HTML Page** item. Name the page "index.html".



Replace everything in this file with the following:

HTML	Copy
<pre><!DOCTYPE html> <html xmlns="http://www.w3.org/1999/xhtml"> <head> <title>Product App</title> </head> <body> <div> <h2>All Products</h2> <ul id="products" /> </div> <div> <h2>Search by ID</h2> <input type="text" id="prodId" size="5" /> <input type="button" value="Search" onclick="find();"/> <p id="product" /> </div> <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.0.3.min.js"> </script> <script> var uri = 'api/products'; </pre>	Copy

```
$(document).ready(function () {
    // Send an AJAX request
    $.getJSON(uri)
        .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
                // Add a list item for the product.
                $('- ', { text: formatItem(item) }).appendTo($('#products'));
            });
        });
    });

    function formatItem(item) {
        return item.Name + ': $' + item.Price;
    }

    function find() {
        var id = $('#prodId').val();
        $.getJSON(uri + '/' + id)
            .done(function (data) {
                $('#product').text(formatItem(data));
            })
            .fail(function (jqXHR, textStatus, err) {
                $('#product').text('Error: ' + err);
            });
    }
}
</script>
</body>
</html>

```

There are several ways to get jQuery. In this example, I used the [Microsoft Ajax CDN](#). You can also download it from <http://jquery.com/>, and the ASP.NET "Web API" project template includes jQuery as well.

Getting a List of Products

To get a list of products, send an HTTP GET request to "/api/products".

The jQuery `getJSON` function sends an AJAX request. The response contains array of JSON objects. The `done` function specifies a callback that is called if the request succeeds. In the callback, we update the DOM with the product information.

HTML

 Copy

```
$(document).ready(function () {
    // Send an AJAX request
    $.getJSON(apiUrl)
        .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
                // Add a list item for the product.
                $('<li>', { text: formatItem(item) }).appendTo($('#products'));
            });
        });
});
```

Getting a Product By ID

To get a product by ID, send an HTTP GET request to "/api/products/*id*", where *id* is the product ID.

JavaScript

 Copy

```
function find() {
    var id = $('#prodId').val();
    $.getJSON(apiUrl + '/' + id)
        .done(function (data) {
            $('#product').text(formatItem(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#product').text('Error: ' + err);
        });
}
```

We still call `getJSON` to send the AJAX request, but this time we put the ID in the request URI. The response from this request is a JSON representation of a single product.

Running the Application

Press F5 to start debugging the application. The web page should look like the following:

The screenshot shows a web browser window with the URL `http://localhost:6497/index.htm` in the address bar. The title bar says "Product App". The main content area has a blue header with the text "All Products". Below it is a list of products:

- Tomato Soup: \$1
- Yo-yo: \$3.75
- Hammer: \$16.99

Below the list is a section titled "Search by ID" with a search input field and a "Search" button.

To get a product by ID, enter the ID and click Search:

The screenshot shows a web browser window with the URL `http://localhost:6497/index.htm` in the address bar. The title bar says "Product App". The main content area has a blue header with the text "All Products". Below it is a list of products:

- Tomato Soup: \$1
- Yo-yo: \$3.75
- Hammer: \$16.99

Below the list is a section titled "Search by ID" with a search input field containing the value "2" and a "Search" button. The output below the search form shows the result: "Yo-yo: \$3.75".

If you enter an invalid ID, the server returns an HTTP error:

All Products

- Tomato Soup: \$1
- Yo-yo: \$3.75
- Hammer: \$16.99

Search by ID

5

Error: Not Found

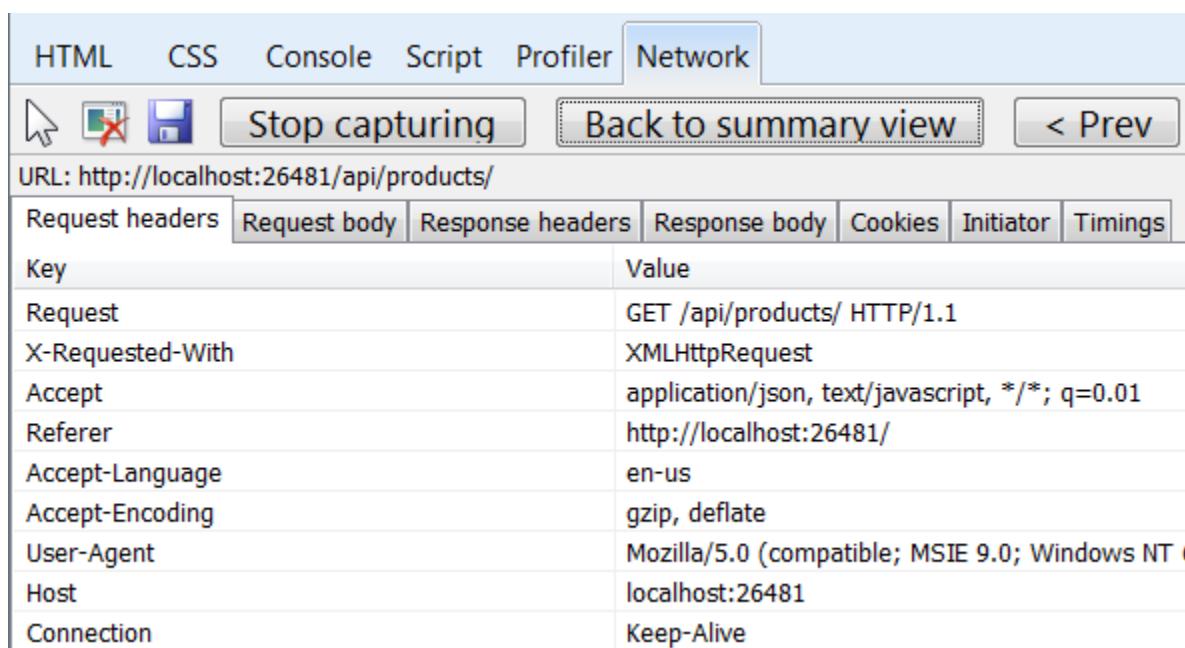
Using F12 to View the HTTP Request and Response

When you are working with an HTTP service, it can be very useful to see the HTTP request and response messages. You can do this by using the F12 developer tools in Internet Explorer 9. From Internet Explorer 9, press **F12** to open the tools. Click the **Network** tab and press **Start Capturing**. Now go back to the web page and press **F5** to reload the web page. Internet Explorer will capture the HTTP traffic between the browser and the web server. The summary view shows all the network traffic for a page:

URL	Method	Result	Type	Received	Taken
http://localhost:26481/	GET	200	text/html	1.66 KB	78 ms
/Content/css?v=ji3nO1pdg6VLv3CV...	GET	200	text/css	0.89 KB	47 ms
/Content/themes/base/css?v=UM62...	GET	200	text/css	24.73 KB	63 ms
/Scripts/js?v=4h5lPNUsLiFoa0vqrItj...	GET	200	text/javascript	325.88...	62 ms
/Scripts/jquery-1.6.2.min.js	GET	304	application/x-ja...	176 B	47 ms
/api/products/	GET	200	application/json	398 B	281 ms

Locate the entry for the relative URI "api/products/". Select this entry and click **Go to**

detailed view. In the detail view, there are tabs to view the request and response headers and bodies. For example, if you click the **Request headers** tab, you can see that the client requested "application/json" in the Accept header.



The screenshot shows the Fiddler application interface. The top navigation bar includes tabs for HTML, CSS, Console, Script, Profiler, and Network, with Network selected. Below the tabs are buttons for Stop capturing, Back to summary view, and navigation arrows. The URL is set to http://localhost:26481/api/products/. The Request headers tab is active, displaying the following table:

Key	Value
Request	GET /api/products/ HTTP/1.1
X-Requested-With	XMLHttpRequest
Accept	application/json, text/javascript, */*; q=0.01
Referer	http://localhost:26481/
Accept-Language	en-us
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; .NET CLR 3.1.4022.15270; .NET CLR 2.0.50727.5475; .NET CLR 3.0.30729.4942; .NET4.0C; .NET4.0E)
Host	localhost:26481
Connection	Keep-Alive

If you click the Response body tab, you can see how the product list was serialized to JSON. Other browsers have similar functionality. Another useful tool is [Fiddler](#), a web debugging proxy. You can use Fiddler to view your HTTP traffic, and also to compose HTTP requests, which gives you full control over the HTTP headers in the request.

See this App Running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by simply clicking the following button.



You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

- [Open an Azure account for free](#) - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- [Activate MSDN subscriber benefits](#) - Your MSDN subscription gives you credits

every month that you can use for paid Azure services.

Next Steps

- For a more complete example of an HTTP service that supports POST, PUT, and DELETE actions and writes to a database, see [Using Web API 2 with Entity Framework 6](#).
 - For more about creating fluid and responsive web applications on top of an HTTP service, see [ASP.NET Single Page Application](#).
 - For information about how to deploy a Visual Studio web project to Azure App Service, see [Create an ASP.NET web app in Azure App Service](#) .
-

Recommended content

[Using Web API with ASP.NET Web Forms - ASP.NET 4.x](#)

Tutorial with code step by step to add Web API to an ASP.NET Forms application for ASP.NET 4.x

[Build RESTful APIs with ASP.NET Web API - ASP.NET 4.x](#)

Hands on lab: Use Web API in ASP.NET 4.x to build a simple REST API for a contact manager application.

[ASP.NET Web API - ASP.NET 4.x](#)

Download ASP.NET MVC 4 and build HTTP services that reach a broad range of clients.

[Enabling CRUD Operations in ASP.NET Web API 1 - ASP.NET 4.x](#)

Tutorial shows how to support CRUD operations in an HTTP service using ASP.NET Web API for ASP.NET 4.x.

Show more ▾

A Beginners Guide to XUnit

How to set up a test project

In-order to create a test, you need to first set up an XUnit project. You should be able to do that in Visual Studio by creating a new project.

Within that project, you can set up a class and create methods within that class.

But how does XUnit know which tests it needs to run? Well you can apply the "Fact" attribute to each method that you wish XUnit to run. XUnit will then know to run this test.

```
1
2      // Test1.cs
3      public class Test1
4      {
5
6          [Fact]
7          public void TestPattern()
8          {
9              var password = "TheBeast";
10
11             Assert.True(IsPasswordValid(password));
12
13             password = "324idfdfdf";
14             Assert.False(IsPasswordValid(password));
15
16             public bool IsPasswordValid(string password)
17             {
18                 return password == "TheBeast";
19             }
20 }
```

As you can see from the above example, I've created two methods. The TestPattern method has the "Fact" attribute assigned to it. Inside that method, there are a number of Assert calls within it. This is where you conduct your tests. XUnit allows you to test on many different things, and here is an example of some of the Assert calls that can be made:

- Contains - Whether a string contains a certain word
- Empty - Whether an IEnumerable is empty
- Equal - Pass in an expected and actual value
- IsNotNull - Pass in an object to see if it has been initialised
- True - Pass in a condition to see if it's true

How each test runs

Even if you have multiple test methods in a test class, each test will always initialise a new instance of the test class. This means that if you wish to run some code before your test commences, you can do so in the constructor.

So what if you want to run some code after a test has progressed? You may wish to log that the test has completed. Well you can inherit the IDisposable interface, and include the Dispose method.

```

1   // Test1.cs
2   public class Test1 : IDisposable
3   {
4       public Test1()
5       {
6           // I can run some code before the test runs
7       }
8
9       [Fact]
10      public void TestPattern()
11      {
12          var password = "TheBeast";
13
14          Assert.True(IsPasswordValid(password));
15          password = "324idfdfdf";
16          Assert.False(IsPasswordValid(password));
17      }
18
19      [Fact]
20      public void TestPattern2()
21      {
22          Assert.True(true);
23      }
24
25      public bool IsPasswordValid(string password)
26      {
27          return password == "TheBeast";
28      }
29
30      public void Dispose()
31      {
32          // I can run some code after the test has finished
33      }

```

30
31
32
33
34

Using the same test for multiple values

If you wish to test multiple values in the same test, rather than creating additional methods to accommodate for this, you can use the "Theory" attribute.

The "Theory" attribute is the same as the "Fact" attribute in the sense that XUnit knows the method is a test. But you have to include additional attributes to a method to allow to pass in multiple values.

One way you can do this is with the "InlineData" attribute. The "InlineData" attribute allows you to pass in an object array with each index representing a parameter in the method. And you can include multiple "InlineData" attributes per method.

In the example below, I've included two "InlineData" attributes. Each "InlineData" attribute has an array with three integers. Each of these integers represent the parameters for the test method in ascending order.

```
1 // Test1.cs
2     public class Test1 : IDisposable
3     {
4         public Test1()
5         {
6             // I can run some code before the test runs
7         }
8
9         [Theory]
10        [InlineData(3, 5, 7)]
11        [InlineData(1, 6, 10)]
12        public void TestPattern2(int i1, int i2, int i3)
13        {
14            Assert.True(i1 <= 3);
15            Assert.True(i2 >= 4);
16            Assert.True(i3 <= 10);
17        }
18
19        public bool IsPasswordValid(string password)
20        {
21            return password == "TheBeast";
22        }
23
24        public void Dispose()
25        {
26            // I can run some code after the test has finished
27        }
28    }
```

25
26
27
28

You can use the "InlineData" attribute, or you can use the "MemberData" and "ClassData" attribute. The "MemberData" attribute allows you to return your parameter data from a method by returning an `IEnumerable<object[]>`. The "ClassData" does the same as "MemberData", but you can return your data in a seperate class and inherit the `IEnumerable<object[]>`.

```
1 // Test1.cs
2     public class Test1 : IDisposable
3     {
4         public Test1()
5         {
6             // I can run some code before the test runs
7         }
8
9         [Theory]
10        [MemberData(nameof(TestData))]
11        public void TestPattern2(int i1, int i2, int i3)
12        {
13            Assert.True(i1 <= 3);
14            Assert.True(i2 >= 4);
15            Assert.True(i3 <= 10);
16
17            [Theory]
18            [ClassData(typeof(ClassTestData))]
19            public void TestPattern3(int i1, int i2, int i3)
20            {
21                Assert.True(i1 <= 4);
22                Assert.True(i2 == 6);
23                Assert.True(i3 == 8);
24
25                public bool IsPasswordValid(string password)
26                {
27                    return password == "TheBeast";
28
29                public static IEnumerable<object[]> TestData()
30                {
31                    return new List<object[]>
32                    {
33                        new object[] { 3, 5, 7 },
34                        new object[] { 1, 6, 10 }
35                    };
36
37                public void Dispose()
38                {
39
```

```
37          // I can run some code after the test has finished
38      }
39  }
40 public class ClassTestData : IEnumerable<object[]>
41 {
42     public IEnumerator<object[]> GetEnumerator()
43     {
44         yield return new object[] { 2, 6, 8 };
45         yield return new object[] { 3, 6, 8 };
46     }
47     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
48 }
49
50
51
52
53
54
55
56
57
```

Walkthrough: Create and run unit tests for managed code

Article • 12/02/2021 • 11 minutes to read •  +10

[Is this page helpful?](#)

In this article

- [Create a project to test](#)
- [Create a unit test project](#)
- [Create the test class](#)
- [Create the first test method](#)
- [Build and run the test](#)
- [Fix your code and rerun your tests](#)
- [Use unit tests to improve your code](#)

See also

This article steps you through creating, running, and customizing a series of unit tests using the Microsoft unit test framework for managed code and Visual Studio **Test Explorer**. You start with a C# project that is under development, create tests that exercise its code, run the tests, and examine the results. Then you change the project code and rerun the tests. If you would like a conceptual overview of these tasks before going through these steps, see [Unit test basics](#).

Create a project to test

1. Open Visual Studio.
2. On the start window, choose **Create a new project**.
3. Search for and select the **C# Console App** project template for .NET Core, and then click **Next**.

Note

If you do not see the **Console App** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message,

choose the **Install more tools and features** link. Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.

4. Name the project **Bank**, and then click **Next**.

Choose either the recommended target framework or .NET 6, and then choose **Create**.

The Bank project is created and displayed in **Solution Explorer** with the *Program.cs* file open in the code editor.

 **Note**

If *Program.cs* is not open in the editor, double-click the file *Program.cs* in **Solution Explorer** to open it.

5. Replace the contents of *Program.cs* with the following C# code that defines a class, *BankAccount*:

C#	 Copy
<pre>using System; namespace BankAccountNS { /// <summary> /// Bank account demo class. /// </summary> public class BankAccount { private readonly string m_customerName; private double m_balance; private BankAccount() { } public BankAccount(string customerName, double balance) { m_customerName = customerName; m_balance = balance; } public string CustomerName { get { return m_customerName; } } } }</pre>	

```
}

public double Balance
{
    get { return m_balance; }
}

public void Debit(double amount)
{
    if (amount > m_balance)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance += amount; // intentionally incorrect code
}

public void Credit(double amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance += amount;
}

public static void Main()
{
    BankAccount ba = new BankAccount("Mr. Bryan Walton",
11.99);

    ba.Credit(5.77);
    ba.Debit(11.22);
    Console.WriteLine("Current balance is ${0}", ba.Balance);
}
}
```

6. Rename the file to *BankAccount.cs* by right-clicking and choosing **Rename** in **Solution Explorer**.
7. On the **Build** menu, click **Build Solution** (or press **Ctrl + SHIFT + B**).

You now have a project with methods you can test. In this article, the tests focus on the `Debit` method. The `Debit` method is called when money is withdrawn from an account.

Create a unit test project

1. On the **File** menu, select **Add > New Project**.

 **Tip**

You can also right-click on the solution in **Solution Explorer** and choose **Add > New Project**.

2. Type **test** in the search box, select **C#** as the language, and then select the **C# MSTest Unit Test Project (.NET Core)** for .NET Core template, and then click **Next**.

 **Note**

In Visual Studio 2019 version 16.9, the MSTest project template is **Unit Test Project**.

3. Name the project **BankTests** and click **Next**.

4. Choose either the recommended target framework or .NET 6, and then choose **Create**.

The **BankTests** project is added to the **Bank** solution.

5. In the **BankTests** project, add a reference to the **Bank** project.

In **Solution Explorer**, select **Dependencies** under the **BankTests** project and then choose **Add Reference** (or **Add Project Reference**) from the right-click menu.

6. In the **Reference Manager** dialog box, expand **Projects**, select **Solution**, and then check the **Bank** item.

7. Choose **OK**.

Create the test class

Create a test class to verify the `BankAccount` class. You can use the `UnitTest1.cs` file that was generated by the project template, but give the file and class more descriptive names.

Rename a file and class

1. To rename the file, in **Solution Explorer**, select the `UnitTest1.cs` file in the `BankTests` project. From the right-click menu, choose **Rename** (or press **F2**), and then rename the file to `BankAccountTests.cs`.
2. To rename the class, position the cursor on `UnitTest1` in the code editor, right-click, and then choose **Rename** (or press **F2**). Type in `BankAccountTests` and then press **Enter**.

The `BankAccountTests.cs` file now contains the following code:

C#	 Copy
<pre>using Microsoft.VisualStudio.TestTools.UnitTesting; namespace BankTests { [TestClass] public class BankAccountTests { [TestMethod] public void TestMethod1() { } } }</pre>	

Add a using statement

Add a **using statement** to the test class to be able to call into the project under test without using fully qualified names. At the top of the class file, add:

C#	 Copy
<pre>using BankAccountNS;</pre>	

Test class requirements

The minimum requirements for a test class are:

- The `[TestClass]` attribute is required on any class that contains unit test methods that you want to run in Test Explorer.
- Each test method that you want Test Explorer to recognize must have the `[TestMethod]` attribute.

You can have other classes in a unit test project that do not have the `[TestClass]` attribute, and you can have other methods in test classes that do not have the `[TestMethod]` attribute. You can call these other classes and methods from your test methods.

Create the first test method

In this procedure, you'll write unit test methods to verify the behavior of the `Debit` method of the `BankAccount` class.

There are at least three behaviors that need to be checked:

- The method throws an [ArgumentOutOfRangeException](#) if the debit amount is greater than the balance.
- The method throws an [ArgumentOutOfRangeException](#) if the debit amount is less than zero.
- If the debit amount is valid, the method subtracts the debit amount from the account balance.

💡 Tip

You can delete the default `TestMethod1` method, because you won't use it in this walkthrough.

To create a test method

The first test verifies that a valid amount (that is, one that is less than the account balance and greater than zero) withdraws the correct amount from the account. Add the following method to that `BankAccountTests` class:

C#

 Copy

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);

    // Act
    account.Debit(debitAmount);

    // Assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited cor-
rectly");
}
```

The method is straightforward: it sets up a new `BankAccount` object with a beginning balance and then withdraws a valid amount. It uses the `Assert.AreEqual` method to verify that the ending balance is as expected. Methods such as `Assert.AreEqual`, `Assert.IsTrue`, and others are frequently used in unit testing. For more conceptual information on writing a unit test, see [Write your tests](#).

Test method requirements

A test method must meet the following requirements:

- It's decorated with the `[TestMethod]` attribute.
- It returns `void`.
- It cannot have parameters.

Build and run the test

1. On the **Build** menu, choose **Build Solution** (or press **Ctrl + SHIFT + B**).
2. If **Test Explorer** is not open, open it by choosing **Test > Windows > Test Explorer** from the top menu bar (or press **Ctrl + E, T**).
3. Choose **Run All** to run the test (or press **Ctrl + R, V**).

While the test is running, the status bar at the top of the **Test Explorer** window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

In this case, the test fails.

4. Select the method in **Test Explorer** to view the details at the bottom of the window.

Fix your code and rerun your tests

The test result contains a message that describes the failure. For the `AreEqual` method, the message displays what was expected and what was actually received. You expected the balance to decrease, but instead it increased by the amount of the withdrawal.

The unit test has uncovered a bug: the amount of the withdrawal is *added* to the account balance when it should be *subtracted*.

Correct the bug

To correct the error, in the `BankAccount.cs` file, replace the line:

```
C#
```

 Copy

```
m_balance += amount;
```

with:

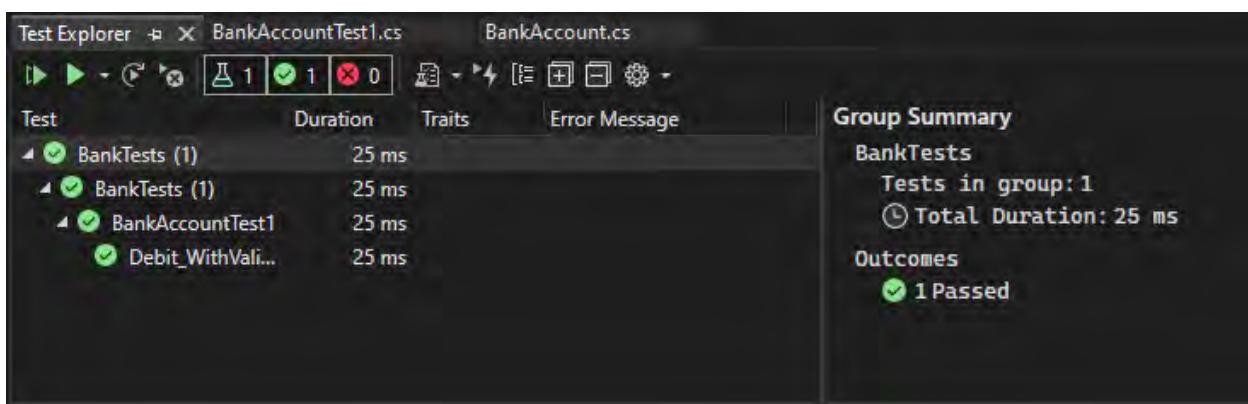
```
C#
```

 Copy

```
m_balance -= amount;
```

Rerun the test

In **Test Explorer**, choose **Run All** to rerun the test (or press **Ctrl + R, V**). The red/green bar turns green to indicate that the test passed.



Use unit tests to improve your code

This section describes how an iterative process of analysis, unit test development, and refactoring can help you make your production code more robust and effective.

Analyze the issues

You've created a test method to confirm that a valid amount is correctly deducted in the `Debit` method. Now, verify that the method throws an [ArgumentOutOfRangeException](#) if the debit amount is either:

- greater than the balance, or
- less than zero.

Create and run new test methods

Create a test method to verify correct behavior when the debit amount is less than zero:

C#	Copy
<pre>[TestMethod] public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException() { // Arrange double beginningBalance = 11.99; }</pre>	

```
    double debitAmount = -100.00;
    BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);

    // Act and assert
    Assert.ThrowsException<System.ArgumentOutOfRangeException>(() => ac-
count.Debit(debitAmount));
}
```

Use the [ThrowsException](#) method to assert that the correct exception has been thrown. This method causes the test to fail unless an [ArgumentOutOfRangeException](#) is thrown. If you temporarily modify the method under test to throw a more generic [ApplicationException](#) when the debit amount is less than zero, the test behaves correctly—that is, it fails.

To test the case when the amount withdrawn is greater than the balance, do the following steps:

1. Create a new test method named

`Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException`.

2. Copy the method body from

`Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException` to the new method.

3. Set the `debitAmount` to a number greater than the balance.

Running the two tests and verify that they pass.

Continue the analysis

The method being tested can be improved further. With the current implementation, we have no way to know which condition (`amount > m_balance` or `amount < 0`) led to the exception being thrown during the test. We just know that an `ArgumentOutOfRangeException` was thrown somewhere in the method. It would be better if we could tell which condition in `BankAccount.Debit` caused the exception to be thrown (`amount > m_balance` or `amount < 0`) so we can be confident that our method is sanity-checking its arguments correctly.

Look at the method being tested (`BankAccount.Debit`) again, and notice that both

conditional statements use an `ArgumentOutOfRangeException` constructor that just takes name of the argument as a parameter:

C#

 Copy

```
throw new ArgumentOutOfRangeException("amount");
```

There is a constructor you can use that reports far richer information:

`ArgumentOutOfRangeException(String, Object, String)` includes the name of the argument, the argument value, and a user-defined message. You can refactor the method under test to use this constructor. Even better, you can use publicly available type members to specify the errors.

Refactor the code under test

First, define two constants for the error messages at class scope. Put these in the class under test, `BankAccount`:

C#

 Copy

```
public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance";  
public const string DebitAmountLessThanZeroMessage = "Debit amount is less than zero";
```

Then, modify the two conditional statements in the `Debit` method:

C#

 Copy

```
if (amount > m_balance)  
{  
    throw new System.ArgumentOutOfRangeException("amount", amount,  
DebitAmountExceedsBalanceMessage);  
}  
  
if (amount < 0)  
{  
    throw new System.ArgumentOutOfRangeException("amount", amount,  
DebitAmountLessThanZeroMessage);  
}
```

Refactor the test methods

Refactor the test methods by removing the call to `Assert.ThrowsException`. Wrap the call to `Debit()` in a `try/catch` block, catch the specific exception that's expected, and verify its associated message. The

`Microsoft.VisualStudio.TestTools.UnitTesting.StringAssert.Contains` method provides the ability to compare two strings.

Now, the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException` might look like this:

```
C# Copy  
  
[TestMethod]  
public void  
Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()  
{  
    // Arrange  
    double beginningBalance = 11.99;  
    double debitAmount = 20.0;  
    BankAccount account = new BankAccount("Mr. Bryan Walton",  
beginningBalance);  
  
    // Act  
    try  
    {  
        account.Debit(debitAmount);  
    }  
    catch (System.ArgumentOutOfRangeException e)  
    {  
        // Assert  
        StringAssert.Contains(e.Message,  
BankAccount.DebitAmountExceedsBalanceMessage);  
    }  
}
```

Retest, rewrite, and reanalyze

Currently, the test method doesn't handle all the cases that it should. If the method under test, the `Debit` method, failed to throw an `ArgumentOutOfRangeException` when the `debitAmount` was larger than the balance (or less than zero), the test method would pass. This is not good, because you want the test method to fail if no exception is thrown.

This is a bug in the test method. To resolve the issue, add an `Assert.Fail` assert at the end of the test method to handle the case where no exception is thrown.

Rerunning the test shows that the test now *fails* if the correct exception is caught. The `catch` block catches the exception, but the method continues to execute and it fails at the new `Assert.Fail` assert. To resolve this problem, add a `return` statement after the `StringAssert` in the `catch` block. Rerunning the test confirms that you've fixed this problem. The final version of the

`Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` looks like this:

C#

 Copy

```
[TestMethod]
public void
Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);

    // Act
    try
    {
        account.Debit(debitAmount);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        // Assert
        StringAssert.Contains(e.Message,
BankAccount.DebitAmountExceedsBalanceMessage);
        return;
    }

    Assert.Fail("The expected exception was not thrown.");
}
```

Conclusion

The improvements to the test code led to more robust and informative test methods. But more importantly, they also improved the code under test.

💡 Tip

This walkthrough uses the Microsoft unit test framework for managed code. **Test Explorer** can also run tests from third-party unit test frameworks that have adapters for **Test Explorer**. For more information, see [Install third-party unit test frameworks](#).

See also

For information about how to run tests from a command line, see [VSTest.Console.exe command-line options](#).

Recommended content

[Unit testing fundamentals - Visual Studio \(Windows\)](#)

Learn how Visual Studio Test Explorer provides a flexible and efficient way to run your unit tests and view their results.

[Assert Class \(Microsoft.VisualStudio.TestTools.UnitTesting\)](#)

A collection of helper classes to test various conditions within unit tests. If the condition being tested is not met, an exception is thrown.

[Assert.IsTrue Method \(Microsoft.VisualStudio.TestTools.UnitTesting\)](#)

Tests whether the specified condition is true and throws an exception if the condition is false.

[Get started with unit testing - Visual Studio \(Windows\)](#)

Use Visual Studio to define and run unit tests to maintain code health, and to find errors and faults before your customers do.

Show more ▾

Application Testing Strategies

- Authentication and authorization: keeping unauthorized users away
- What about authenticated users?

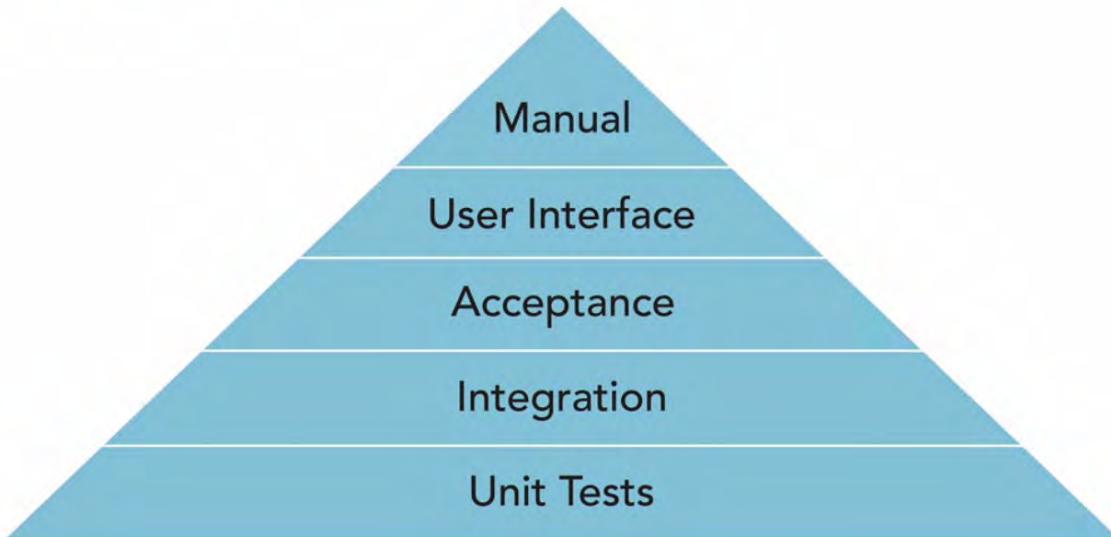
When we think about security in our Apps the first thing that comes to mind is Authentication and Authorization. So we make sure that we keep unauthorized users away from our app. And that's a great practice but there is more to building robust applications. There might also be authenticated users with good intentions that might break our app. But how is that possible?

Bugs, Bugs, Bugs...

- Edge use cases
- Invalid form data

When we as developers write code, a lot of times we write code that does not take into consideration all the use cases. For example when creating forms, we assume that users will enter the right data, but it's common for the users to not use our apps the way we expect them to do. For example in a field where you expect them to write a positive number they might enter a negative one. All these unpredicted cases introduce Bugs to our apps and Buggy apps create angry customers and angry customers create a lot of headache and lower revenue. So what should we do to make sure that we develop stable apps and which are the core strategies for doing so.

Pyramid of Testing



The Testing Pyramid or the Testing Triangle is a strategy guide for implementing software testing, which is a strategy used to make sure that our app works as expected before we deployed to production. In this Pyramid we have five levels starting from the bottom to the top. We start with Unit Testing a Unit Testing is testing a single unit of an application, a unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In Object Oriented Programming the smallest unit is a method. The next level is the Integration Test, integration test is a level of software testing where you divisional units are combined and tested as a group. The purpose of this level of testing is to expose fault in the interaction between integrated units. Here we can test if two classes or two modules interact as expected with each other. Acceptance Test is another level and the purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it's acceptable for delivery. The purpose of acceptance tests is to evaluate the system's compliance with the business requirements and assess whether it's acceptable for delivering. The highest level of automated tests are the UI Tests, these tests actually drive the Web Browser in an automated fashion performing mouse clicks typing text and clicking links. The UI Tests are great to check if the app works as expected. So for example when the user clicks a link, When the user clicks on a button, when the user types a text etc, but the automated tests cannot tell you what the user experience of your website is like. That's why it's important to get a real variety of typical users to look at your website. And for that we have the Manual Tests, the problems that manual testers are able to find are different from the ones you will find using Unit Integration or any other tests.

Unit testing is the base of a testing pyramid which means it's also really important for the other testing hierarchies. A strong base in a pyramid is so that the pyramid will also be strong and stable. The purpose of unit testing is to isolate each part or unit of our app and test that part if it works as expected. But why use unit testing and which are the main benefits of doing so? When you write an app you don't know from the top of your head how the app will turn out to be in the near future, you might want to add new

features or remove existing ones, but changing the structure of your app has its own reasons. You might break existing features by changing the existing code. But having unit testing means that we know exactly what code we broke with our changes, so we can fix the issues without deploying the new version of our app. So having unit tests means that we can easily refactor our code. Also, when you write unit tests, you need to make sure that the single method is used for a single purpose. So this way we create modularity in our code which is then easy to reuse and this is the second benefit of unit testing, reusable code. Without unit testing when you write new code we need to test it before we deploy our code to production. And to do so, we normally set break points and then we check if the results are what we expect them to be, but writing unit tests is easier than debugging the code every time. We write a unit test, we create a method in our app. We test it and then, deploy it. So, with unit testing, we can develop faster. With unit testing, it's also easier to debug because if we break our code the unit test will fail and this way we can easily locate where the issue is. All these together reduce the development costs. Which are some tips that we need to keep in mind when we write unit tests? When unit testing there are some tips that we need to keep in mind. One of them is to not unit test everything. An app might have thousands of methods and some of them might not even be complicated at all. So instead of unit testing everything, we need to focus on the tests that impact the behavior of the system. The data that we use to unit test, if not the same as in production, needs to be close to production data. Because this way, you know when you fix and test an issue in your development environment it's also fixed in production, when writing unit tests, you need to make sure that unit tests are not dependent in each other because if a unit test fails, it should not cause the failure of other unit tests. And last, but not least, perform unit tests continuously and frequently. What this means is that when you add a new feature you need to add new unit tests. But how can we perform or integrate unit testing into our ASP.NET Core apps? We can use a lot of frameworks that are available but the main ones are the MSTest, NUnit, and the xUnit.net framework. And you'll learn more about these frameworks in the upcoming parts.

MSTest is Microsoft's official unit testing framework, and it comes by default with Visual Studio 2005 and later. The main benefit of MSTest is that the unit tests written using this framework can be run directly from Visual Studio or by using the MSTest.exe from command line. But which are the main elements of MSTest? When you want to define a C# class as a unit testing class, you need to use the test class decorator. And inside this class you can have helper methods, which you don't want a unit test, for example, you want to return a list of data, and you can also have unit testing methods. To identify methods that contain unit testing code, you use the test method decorator. And inside this method, when you want to check for the result, you need to use the assert method. The assert method will test the condition or behavior against an expected result. The MSTest also offers initialization and cleanup methods, which are used to prepare unit tests before running and cleaning after execution. So, an example would be when you want to create a reference to the database and then, once the tests have run successfully, you want to clean up this connection.

NUnit is a unit testing framework for all .NET languages, but its existence dates back almost 30 years long before .NET was even a thing. If you have previously worked with Java, there is a chance you have heard of the JUnit which is a unit testing framework for Java applications. NUnit is derived from JUnit

and is used for unit testing .NET applications. The NUnit runs very well with the .NET programming languages like C#, Visual Basic .NET, et cetera. It's open source and can be used with a graphical user interface or a command line, but which are the core NUnit elements? When you want to identify the classes that contain the test methods, you can use the SetUp decorator and when you want to identify the methods that contain the test code, you use the Test decorator. When you want to provide different inputs to the same test method, then you use the TestCase, and if the input that you want to provide to a test method comes from an external file, which can be a CSV file or from a database, then you use the TestCaseSource. At the end we have the TearDown decorator, which is the opposite of the SetUp decorator and is used to clean up the resources after the tests have run successfully.

The xUnit framework is the latest technology for testing .NET applications written on top of the .NET programming languages like C#, F#, VB.NET, et cetera. It was also written by the original inventor of NUnit. One of the core benefits of using the xUnit.net is that it runs really well with the .NET programming languages like C#, VB.NET, et cetera. It has an intuitive terminology and excellent extensibility, because this framework works really well with other tools like ReSharper, Calderash, and Summary. But which are the core xUnit.net elements? The two most important decorators are the fact and theory. We use the fact attribute when we want to unit test a method which doesn't take any parameters. And when we want to pass some data to our unit testing method, then we use theory. To provide data to theory you can use the data attributes, like inline data, member data, and class data. And we are going to talk about these attributes in more details in the upcoming chapter.

So far, we have talked about the MSTest, the NUnit, and the xUnit.net frameworks as three possible unit testing frameworks that we can use throughout this course. But now, it's time to choose one of them. Well, let's compare them with each other and decide by the end of this video. To set up a unit testing class, in MSTest you need to use the test class decorator. In NUnit you need to use the test fixture, and in xUnit.net you don't need to write anything. If you want to define a simple unit testing method, which doesn't take any parameters, in MSTest you use the test method, in NUnit you use test, and in xUnit.net, you use the fact attribute. When you want to set up any resources in the unit testing file, like a database reference or a service reference, in MSTest you use the test initialize, in NUnit you use the setup attribute, but in xUnit you don't use any attributes, you can just use the constructor of that class. And when you want to clean up the resources, in MSTest you use the test cleanup, in NUnit the tear down, and in xUnit you can just use the dispose method. When you want to ignore a unit testing method because you don't want to use that method anymore, in MSTest you use the ignore attribute, in Nunit you use the ignore attribute and you can pass as a parameter the reason why you are ignoring that method, and in xUnit you use the fact, but you pass as a parameter a skip property, why you want to skip or ignore that method. Now, when you need to pass data to a unit testing method, in MSTest you use the data source, in NUnit you use the theory, and in xUnit you use the theory attribute, as well. But in xUnit you can then use the inline data, member data, and class data attributes based on the type of data that you want to pass to your method. Well, we can see that they all have their advantages. But we are going to move forward with xUnit. It's the latest technology when it comes to testing dot net languages, works well with other tools, it's open source, and community-focused.

Console Applications

In this chapter, we are going to learn how to unit test an existing ASP.NET Core console application. I'll be using the .NET Core 3.1, which is current latest stable version, but you can use any version of .NET Core, 2.2 or later. Let's go to Visual Studio and see our project. This is a simple console application named Calculator App, and inside here, you are going to see the default program.cs file, which has the main method, also known as the entry point in the .NET applications. And we have another file, the MathHelper.cs. Inside this file, which is a C# clause, you'll see a couple of methods. The first one is the IsEven method. This method checks if a number is an even number or not, and it returns true or false. The other method, named Diff, finds the difference between the second parameter and the first one. The other one, of the Add method, returns the sum of two parameters. Then next, we have the Sum and the Average methods. The Sum method will return the sum of all the elements in an integer array. And the Average one will return the average of the numbers in an integer array. Then next, we have a class member, which is going to return a list of objects. We are going to use this member when we want to pass as parameters for our testing methods a membered data. Then at the ends, we have the GetEnumerator method, which is used to return a list of objects when we want a test as a parameter to our unit testing method our class.

Now that we know what our app looks like, it's time to set up our testing project. So for that, let's go to a Visual Studio. In here we are going to create a new project. So, let us right click on B solution, then go to add new project, search for xUnit, and choose the xUnit test project, the.net core version. Then click the Next button. Let us name our testing project. I'll name this project calculator app, and then dot test. So, we know that this is the unit testing project. Click the create button to create these projects. Before we write any code, let us right click and add a reference to our calculator app project because there's the project that we want to unit test. So, check the check box and then click the OK button. If we expand the dependencies inside the projects, we're going to see our calculator app. So, let us close the dependencies. We see that when we created the unit testing project, a unit testing class and a unit testing method were created by default, and we can see that the unit testing method has the fact attribute, which means that this is a unit testing method without par meters. If you want to pass par meters to unit testing method, you need to use the theory attribute, and we're going to talk about that in the next part. So, let us change the name of this class. So, right click and then go to rename, then I'll say in this class Mathhelptest, and then press Enter. Yes, we want to change the name inside the file as well. So, here we have the math helper test. Now, let us write our first unit test. Let us change the name of this method to IsEven Test, because we want to test the IsEven method. Inside here let us create a reference to our class further down just write var calculator is equal to new math formulas. Then inside here I'll define two variables, so, int X is equal to one and int Y is equal to two. Let us now get the results for the X and the Y for that let's write var X result is equal to calculator.IsEven. We passes a par meter to the X. And then var Y result is equal to calculator.IsEven, and we passes a par meter the Y. Now it's time for us to check the results, and to check the results we are going to use the assert class. So, let's write in your assert. And assert provides multiple methods. Some of these methods are the equal, when we want to compare two objects or two strings, the true when we want to check the resolved is true, not now then we want to check if the result is not now and false when we want to

check if the result is false. Now for the X, we know that X is not an even number because one is an odd number and Y is an even number. So, let us check if the assertion is false for X result, which is true because X is not an even number and assert.true for the Y result. Let us save the changes, let us build our solution. Let us go to test, then test Explorer to run our test. So, we can see that now in here we have the calculator.test project, and inside here we have the Mathhelper.testmethod. Let us right click and run. So, we see that our test best successfully. If we go and change the asserting here from false to true, which means that the X resolved is a true value, it will fail. So, let us see the changes go back to test, test Explorer, and run the test one more time. Now, we see that the unit test fails, and the reason why it fails is because let's drag this to the right. It fails in line nine. And if we just move this from here, here we have the line nine. So, this is the method that it's failing and this method is failing in line 19. Here we can see that it failed in line 19 and the reason for that is because it expected a true value. So, we wrote in your truth, but the result was a false. So, let us close this window change these back to false, and save the changes.

Using the effect attribute when writing unit tests means that the unit test method does not have any parameters. And whatever data you need to test, you need to provide them inside the method. But what if you want to test a lot of data? In that case, you need to define a lot of variables and this way the method will get bigger and bigger, which is not a good idea when it comes to writing clean unit tests. A VX unit .NET Framework provides the Theory attribute for parameterized tests, and we pass the parameters using different attributes likeInlineData, MemberData, and ClassData. So, let us start with theInlineData. For that, let's go to Visual Studio. In here after the IsEven method, we are going to create another one. So let us write in here, public void DiffTest, 'cause we want to test the diff method, and this will take three parameters, the int x, int y and the int expectedResult. Now inside here, let's write var calculator is equal to new MathFormulas, and let's write in here var result is equal to calculator.Diff, which takes two parameters, the X and Y. And now it's time to assert the results. So Assert.Equal, and we want to check if the result was the same as the expected result, and that's it. But we have not provided any values for the X, Y and the expected value. And we are going to do that by using the Theory attribute. So, Theory, we know that this method now expects some value, and it says in here that the Theory method must have test data, and we are going to provide the test data by using theInlineData attribute. So,InlineData, the X is going to be one, it's going to be two, and the difference is one. So this is the expected result. This means that the X will have the value one, two will be the value of Y, and the expected value will be one. Now, if you want to have two parameters, you can remove the expected value, then you need to remove this one from here and as you can see you are already getting an error, which says that there is no matching method parameter for value one, so remove this one. And then you need to manually provide the expected value, but it's better to have it as a parameter in our methods. I'll just revert these changes. Then I'll save the changes and go to Test, then Test Explorer. Inside here we are going to see our new method, which is the DiffTest method, and let's run this method. So we see that the method ran successfully. Now, if you want to provide more data, you can use anotherInlineData attribute in the same method. So I'll just copy this line and paste it down here. I'll just change this to three and leave these two on so I know it will fail. Now let us go to Test, then Test Explorer, and here now, we can see that we have two tests. The second one has not run yet, so let us right-click and then click the Run option. So we see that the second one failed, and the reason was because it was expecting a two value, but the actual value was one. So let us change the value to

two, and now the test will succeed. Now, let us unit test the Sum method. I'll just create the Theory first, and then just write in here public void SumTest, because we are testing the Sum method, and as a parameter, we'll take a list of numbers, so I'll just write in here int values and then it has an int expectedValue. So let's write in here var calculator is equal to new MathFormulas, and then in here write var result is equal to calculator.Sum, and we pass as the parameters the values. And now let us Assert the result, so Assert.Equal, the result, and the expected result. Now we see that we have an error in here and that's right because we have defined this to be a Theory, but we have not provided any data, so let's write in hereInlineData. The first parameter is going to be a list of numbers, so I'll just write in here new int, I'll have three numbers that are going to be one, two, three, and the result is going to be six. Let's create another one, it has three elements. The first value is going to be minus four, minus six, minus 10 and the result is going to be minus 20. Let us save the changes and go to Test, Test Explorer and here now we can see we have the SumTest unit test. So let us right click and then run. We see that both cases passed.

I asked you to unit test the add and average methods using the inline data attribute to pass parameters to these methods. Let us now walk through my solution together. For that, let's go to Visual Studio. And you scroll down to the DiffTest, because the AddTest is similar to the DiffTest. I'll just copy this unit test and paste it down here. Then change the name to AddTest, leave the parameters the same and then change the Diff method in here to Add method. Now, of course we need to change the inline data parameters because one plus two, will result in three. Let us now also use some negative numbers. So for example, minus four, minus six, the result is going to be minus 10. And save the changes. Let us now create a unit test for the average method. For that, I'll just copy the SumTest. And then paste it down here, I'll change the name from SumTest to AverageTest. Leave the parameters the same. Then change the method from Sum to Average. Now, let us change the inline data parameters. So for example, the average of one, two and three is two. And the average of the other numbers or let us write in here, two, so we know that it will fail. Now let's go to Test. Go to Test Explorer. And here now we have nine unit tests. Let us run all of them. So right click and then choose Run. And we can see that from all the unit tests, one failed. Let us expand the unit test in here, we see that the average test failed and from them, the one that has the expected value set to two. And the reason is that the expected value is minus six, but the actual provided value was two. So, let us go and change the two to minus six. Let us save the changes and run the test one more time. So right click, and then Run. So you can see that all the tests now passed. Let us go to the AddTest method. So I'll just close this window, then scroll up, right click, go to Definition. Now instead of returning the sum of x and y, I'll just write some code, which will break this method. So plus one. Let us save the changes. Let's go to Test, Test Explorer, let us run the unit test. So we can see now that the AddTest just failed. If we go in here, we'll see that even though we use the Add method, and one plus two is equal to three, because that simple and we know that, the unit test still failed. So this means that something is wrong with our method. So let us go to our method and revert the change. So this is how you easily find bugs when you have unit test in your code. So for example, if you have an existing method, then you modify the method. Now the unit test fails, you know that you broke something in that method. So you go and fix your method. Then I'll save the changes. So let us run the test one more time. We then go to Test and then Run All Tests. We can now see that all tests passed successfully.

The [MemberData] attribute can be used to fetch data from a static property or method.

Using in-line data to paste parameters to unit testing method is great. And you can also use multiple in-line data attributes to paste more sets of data. But what happens if you want to paste like 50 plus sets of data? The unit testing method becomes really long, and it will be hard to manage. But the xunit dotnet framework has a solution for this. Instead of loading all the data using the in-line attribute, you can load them from a method or a property using the MemberData. Well, let's go to Visual Studio and see this in action. When you scroll down to the bottom of the file and let us create a Theory. Then in here write public void Add because we are going to test the Add method but to distinguish these two methods from each other, I'll write in here Add MemberData underscore Test. This will take three parameters the same like the Add data. So I'll just scroll up to the AddTest method, copy these three values, scroll down, paste them in here, then scroll up, copy the rest, and scroll down and paste them inside the methods body. Now we get an error because we defined in here the parameters, but we didn't provide any data. But instead of using the in-line data, we are going to use the MemberData. Now if we go to our mathhelper.cs file, and scroll down in here, you'll see that we have a property named Data, which returns a list of objects. Now in here, we see that we have only four objects, but we can have way more than four. So for example, we can even get all this data from the database and test the data. So let us go back to our test file, and then in here write instead of in-line data, we are going to use the MemberData, so MemberData. The first parameter that we need to provide is the member name, which is the Data member, or we can just write in here nameof and then here write MathFormulas.Data. Then as a second parameter, I'll provide the member type. So MemberType is equal to type of MathFormulas. Let us save the changes. Go to Test, Test Explorer, expand the test and here you will see the Add MemberData Test. So let us right-click and then click Run. You'll see in here that the unit test passed. And in here in parentheses, you will see the number four, and that's because we passed using the data attribute four objects. So let us close this one. Let's close the other window too. So we had the same method where we check the results and the expectedResult. In your last modified the Equal method, because the first parameter needs to be the expectedResult, and the second parameter needs to be the result. So let us modify the other unit test too, I'll just copy this value, change the name in here and then change it in here then scroll up. I'll change it to be camel case, so we have the rest camel case. Now let us scroll down, let's copy these value and paste them in all of the Equal methods. Let us now save the changes, go to Test, then Test Explorer, right click and run all the tests. we see that everything worked as expected.

You learned how to use the member data attribute to pass parameters to a unit testing method, but member data required aesthetic field, a property, or a method that returns an IEnumerable of an object array, but if you don't want to create aesthetic field, property, or a method to pass data, then you can inherit from the IEnumerable object array in your class, and instead of passing member data, you can then pass the class data using the class data attribute to your unit testing method. So let us go to Visual Studio and see this in action. Before we write any code, let us go to the MathHelper.cs file, and in here we see that the MathFormulas class inherits from the IEnumerable object array, and if you scroll

down, we have the implementation of `IEnumerable` to get the `Enumerator`, so we can pass a list of data to our unit testing method, and it just returns a list of new object arrays. So let us go to our `MathHelperTest` file, and you scroll to the bottom, and we are going to start by creating a `Theory`, then instead of using the member data we are going to use the `ClassData` attributes, so `ClassData`, and here we are just going to define a `VClass` name. So let's write in here `typeof`, and the class name is `MathFormulas`. As you can see here in the `MemberData`, we had to provide the field name, but it can be a method name, or it can be a property name. And you now let us create our method, so I'll just write in here `public void`, we are going to test the `Add` method, but to distinguish it from the other methods, I'll just write `Add` and then `ClassDasta_Test`. This will also take three parameters, so I'll just copy all this code, remove the parenthesis, and paste it in here. So we have the `expectedValue` first, and then the result. Let us save the changes, go to `Test`, then `Test Explorer`, and here we are going to see our new unit test, so let us go to the `Add Class Test`, right-click, and then `Run`. We can see that everything worked as expected. If you want to change this test, you can just go your class, and here let us remove the last two tests, change the three in here to four, save the changes, go back to `Test`, then `Test Explorer`. Let us right-click and then `Run`. Let us right-click and `Run` again because it needed some time for the changes to be reflected in here. Now we can see that the first one passed, but the second one failed because the expected result was four and the actual value was three. So if we go to the `MathHelper`, we see that it was expecting a four, but the result was actually a three. Now before we move to the next chapter, let us learn how to skip a test. We just need to modify the `Theory` attribute. So in here, I'll just write inside parenthesis, I'll just write `Skip`, and then we need to provide a skipping reason. So I'll just write in here, this is just a reason. Let us save the changes, then go to `Test`, and `Run All Tests`. Now in here you can see that 13 passed and one unit test was skipped, and that is the `ClassData_Test`.

Introduction to Unit Testing

Overview

In this lab, you'll learn about unit testing. Unit tests gives you an efficient way to look for logic errors in the methods of your classes. Unit testing has the greatest effect when it's an integral part of your software development workflow. As soon as you write a function or other block of application code, you can create unit tests that verify the behavior of the code in response to standard, boundary, and incorrect cases of input data, and that verify any explicit or implicit assumptions made by the code. In a software development practice known as test-driven development, you create the unit tests before you write the code, so you use the unit tests as both design documentation and functional specifications of the functionality.

Visual Studio has robust support for unit testing, and also supports deep integration with third-party testing tools. In addition, you can leverage the power of Visual Studio Online to manage your projects and run automated tests on your team's behalf.

Objectives

In this hands-on lab, you will learn how to:

- Create unit tests for class libraries
- Create user applications that are highly testable
- Take advantage of advanced Visual Studio features, such as data-driven testing, code coverage analysis, and Microsoft Fakes
- Create a continuous integration environment that automatically runs unit tests upon file check ins

Prerequisites

The following is required to complete this hands-on lab:

- [Microsoft Visual Studio 2013](#) (with Update 2 RC applied)
- [A Visual Studio Online account](#) (only required for exercise 4)

Notes

Estimated time to complete this lab: **60** minutes.

Note: You can log into the virtual machine with user name “**User**” and password “**P2ssw0rd**”.

Note: This lab may make references to code and other assets that are needed to complete the exercises. You can find these assets on the desktop in a folder named **TechEd 2014**. Within that folder, you will find additional folders that match the name of the lab you are working on.

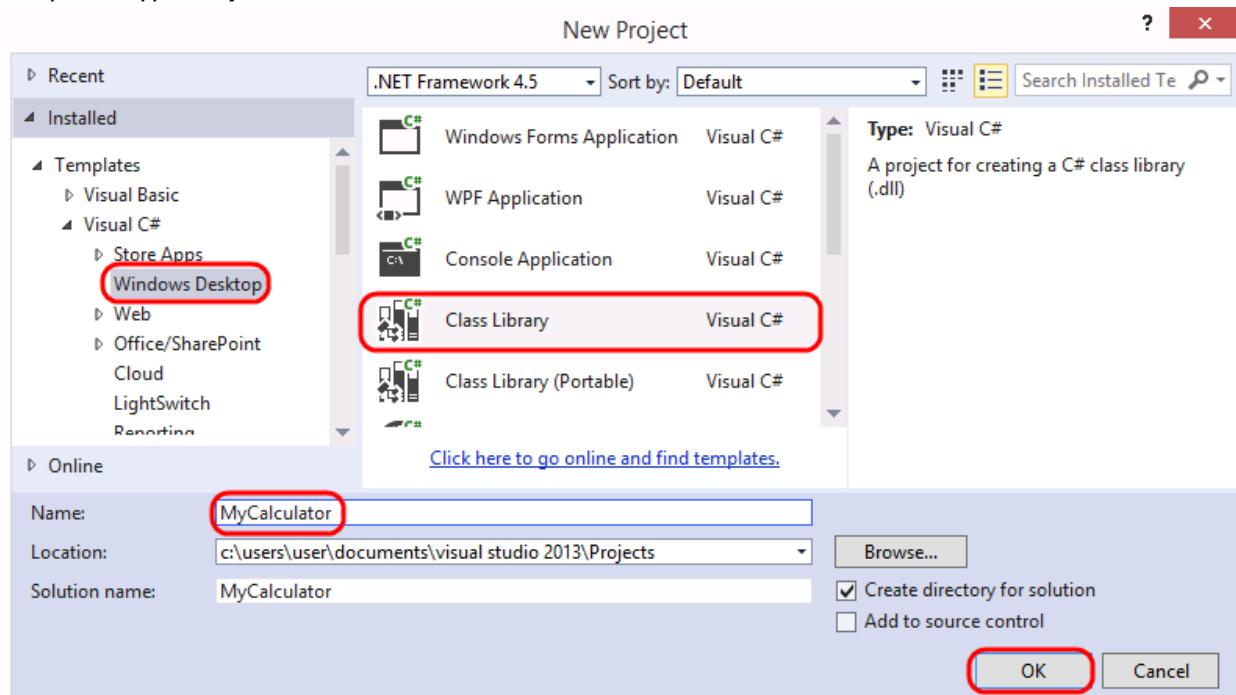
[Exercise 1: Creating a project and supporting unit tests](#)

In this exercise, you'll go through the process of creating a new project, as well as some supporting unit tests.

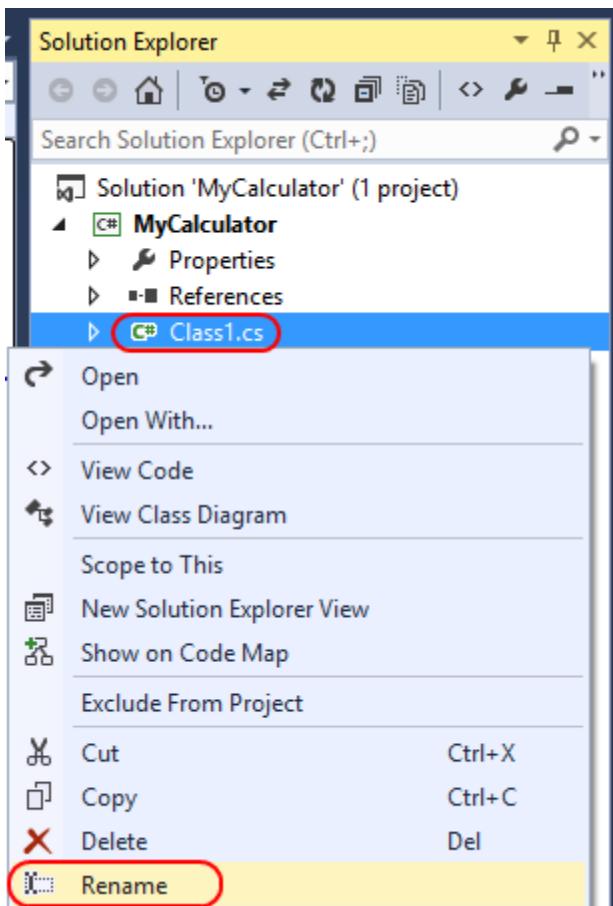
Task 1: Creating a new library

In this task, you'll create a basic calculator library.

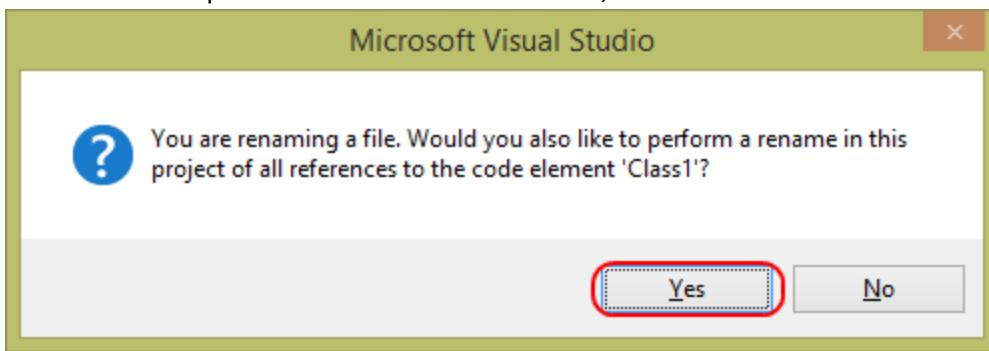
1. Open **Visual Studio 2013**.
2. From the main menu, select **File | New | Project**.
3. In the **New Project** dialog, select the **Visual C# | Windows Desktop** category and the **Class Library** template. Type “**MyCalculator**” as the **Name** and click **OK**.



4. In **Solution Explorer**, Right-click the **Class1.cs** and select **Rename**. Change the name to “**Calculator.cs**”.



5. When asked to update the name of the class itself, click **Yes**.



6. Add the following **Add** method to **Calculator.cs**.

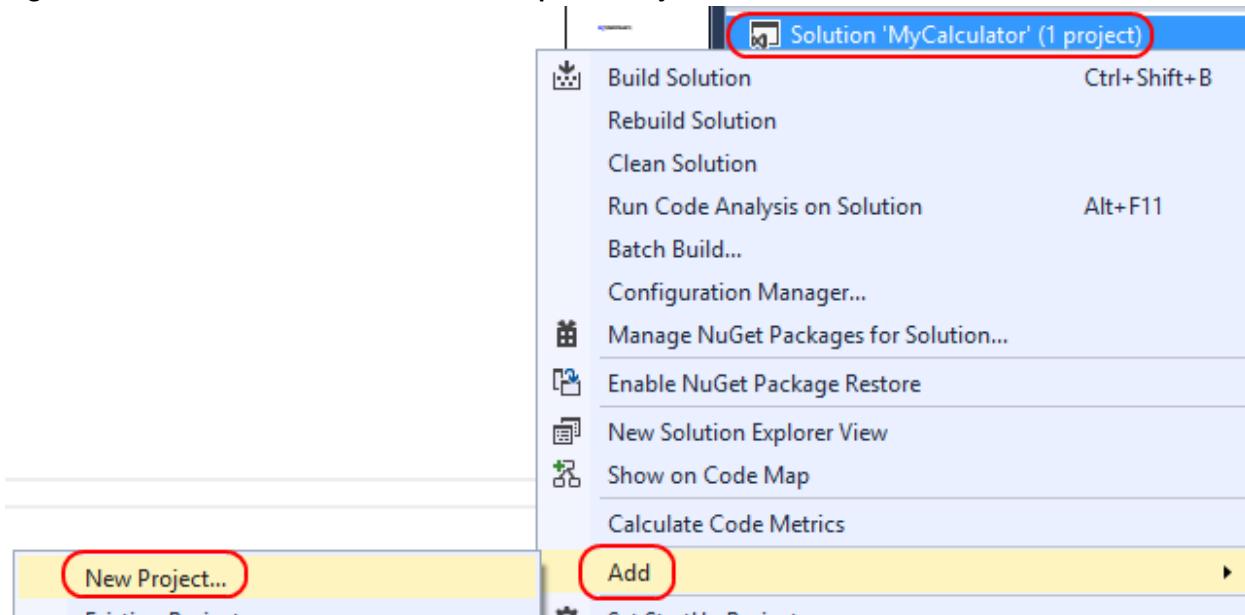
```
C#
public int Add(int first, int second)
{
    return first + second;
}
```

Note: For the purposes of this lab, all operations will be performed using the `int` value type. In the real world, calculators would be expected to scale to a much greater level of precision. However, the requirements have been relaxed here in order to focus on the unit testing.

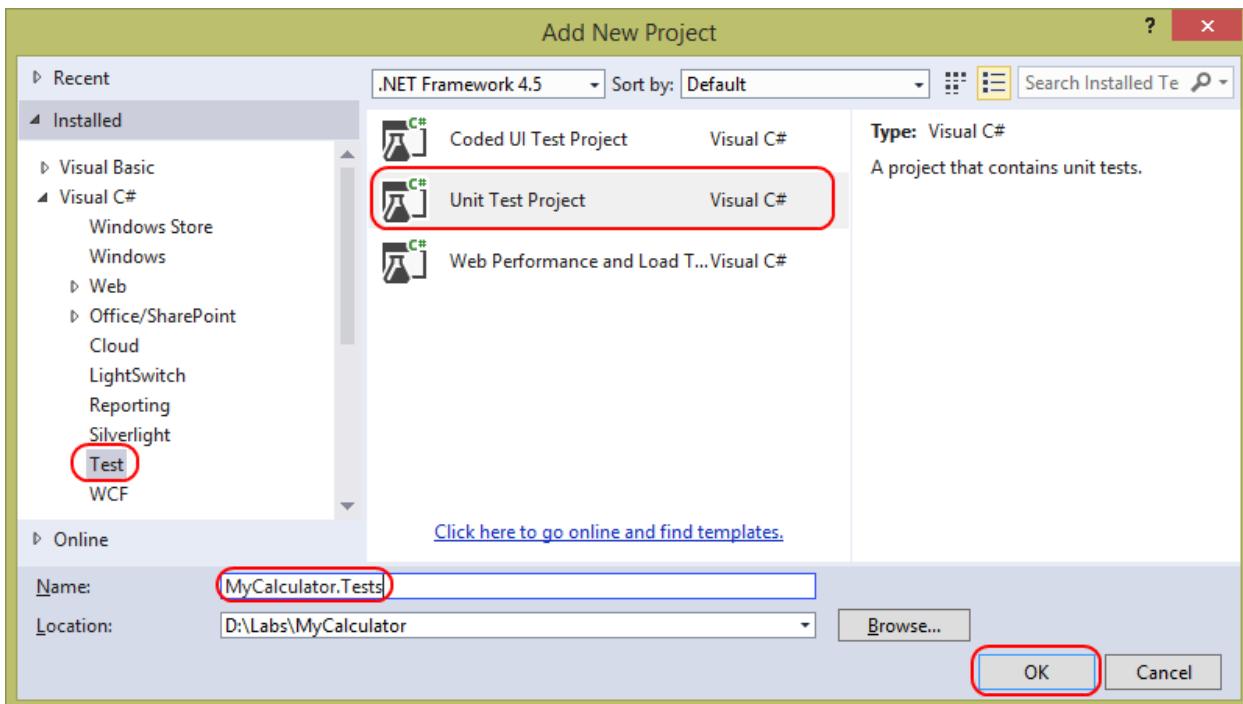
Task 2: Creating a unit test project

In this task, you'll create a new unit test project for your calculator library. Unit tests are kept in their own class libraries, so you'll need to add one to the solution.

1. Right-click the solution node and select **Add | New Project....**



2. Select the **Visual C# | Test** category and the **Unit Test Project** template. Type "**MyCalculator.Tests**" as the **Name** and click **OK**. It's a common practice to name the unit test assembly by adding a ".Tests" namespace to the name of the assembly it targets.

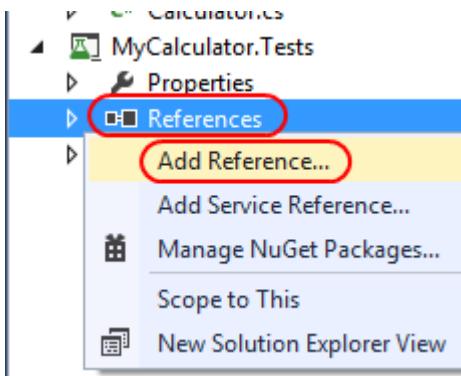


The wizard will end by opening the default unit test file. There are three key aspects of this class to notice. First, it includes a reference to Visual Studio's unit testing framework. This namespace includes key attributes and classes, such as the **Assert** class that performs value testing. Second, the class itself is attributed with **TestClass**, which is required by the framework to detect classes containing tests after build. Finally, the test method is attributed with **TestMethod** to indicate that it should be run as a test. Any method that is not attributed with this will be ignored by the framework, so it's important to pay attention to which methods do and don't have this attribute.

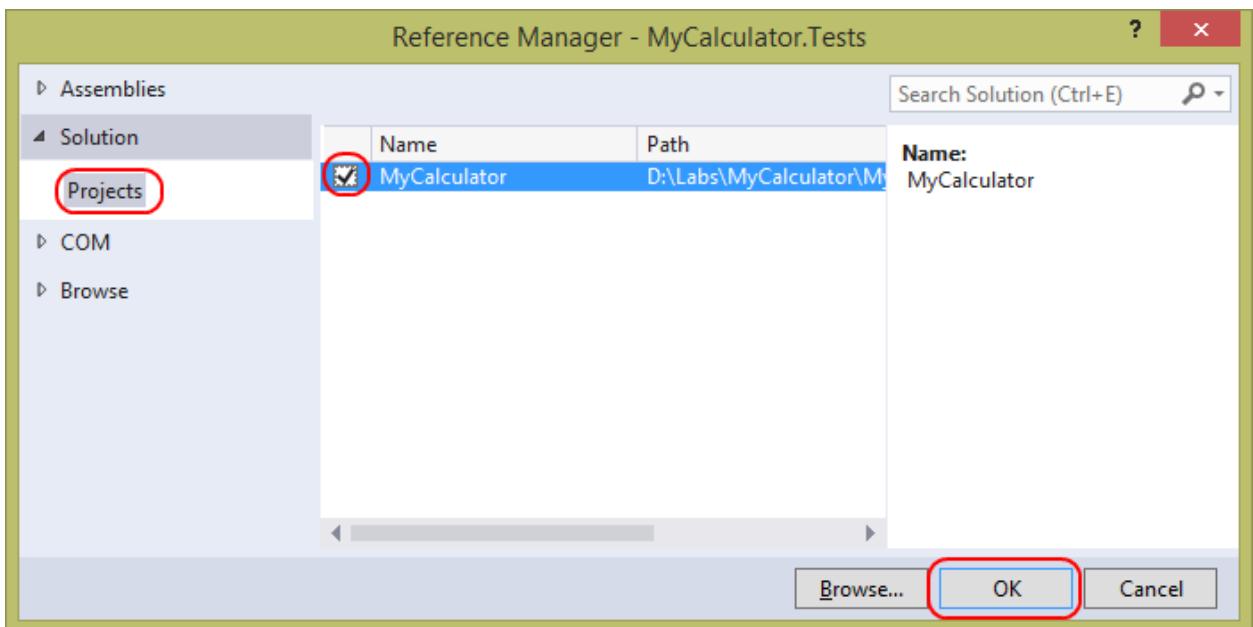
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MyCalculator.Tests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

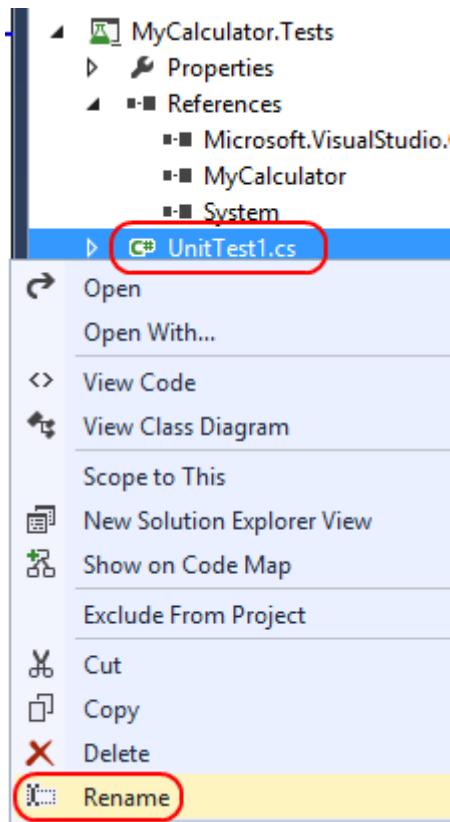
3. The first thing you'll need to do with the unit test project is to add a reference to the project you'll be testing. In **Solution Explorer**, right-click the **References** node of **MyCalculator.Tests** and select **Add Reference....**



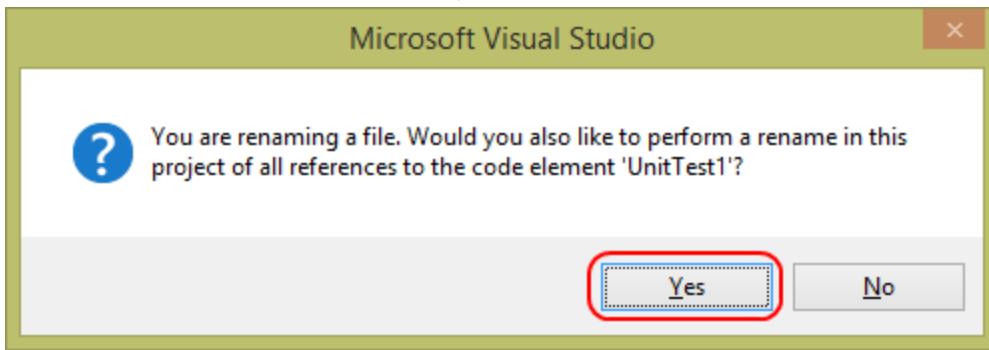
4. Select the **Projects** category in the left pane and check **MyCalculator** in the main pane. Click **OK** to add the reference.



5. To make the project easier to manage, you should rename the default unit test file. In **Solution Explorer**, right-click **UnitTest1.cs** and rename it to “**CalculatorTests.cs**”.



- When asked to rename the class itself, click **Yes**.



- At the top of **CalculatorTests.cs**, add the following **using** directive.

```
C#
using MyCalculator;
```

- Rename **TestMethod1** to **AddSimple**. As a project grows, you'll often find yourself reviewing a long list of tests, so it's a good practice to give the tests descriptive names. It's also helpful to prefix the names of similar tests with the same string so that tests like **AddSimple**, **AddWithException**, **AddNegative**, etc, all show up together in various views.

```
C#
public void AddSimple()
```

Now you're ready to write the body of your first unit test. The most common pattern for writing unit tests is called **AAA**. This stands for **Arrange, Act, & Assert**. First, initialize the environment. Second, perform the target action. Third, assert that the action resulted the way you intended.

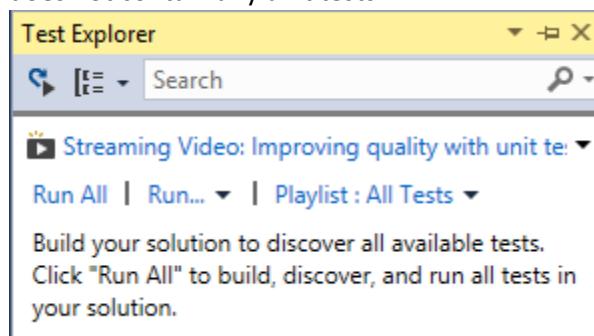
9. Add the following code to the body of the **AddSimple** method.

C#

```
Calculator calculator = new Calculator();
int sum = calculator.Add(1, 2);
Assert.AreEqual(0, sum);
```

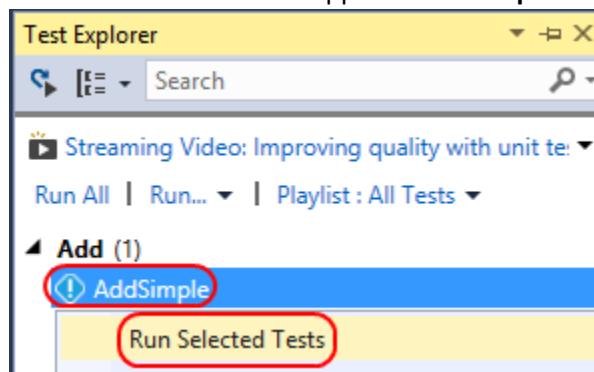
This test has only three lines, and each line maps to one of the **AAA** steps. First, the **Calculator** is created. Second, the **Add** method is called. Third, the **sum** is compared with the expected result. Note that you're designing it to fail at first because the value of **0** is not what the correct result should be. However, it's a common practice to fail a test first to ensure that the test is valid, and then update it to the expected behavior for success. This helps avoid false positives.

10. From the main menu, select **Test | Windows | Test Explorer**. This will bring up the **Test Explorer**, which acts as a hub for test activity. Note that it will be empty at first because the most recent build does not contain any unit tests.

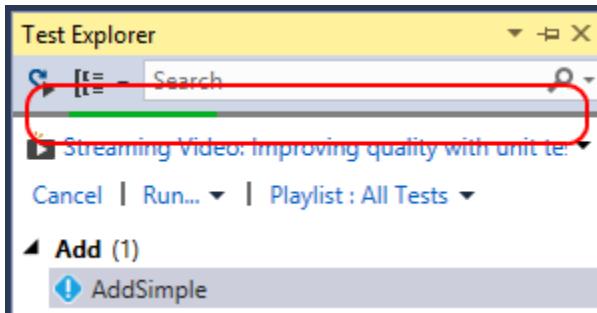


11. From the main menu, select **Build | Build Solution**.

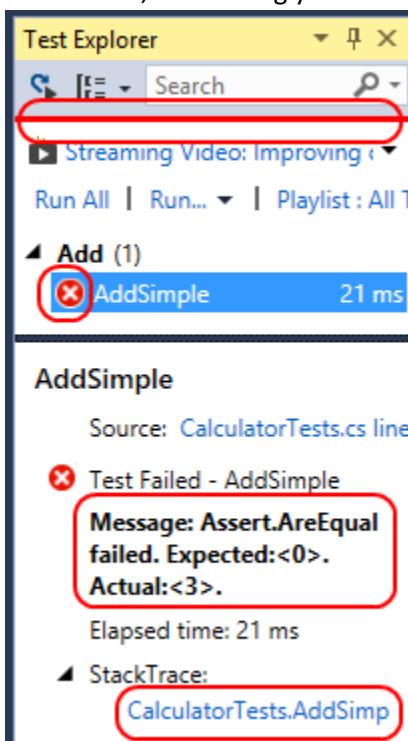
12. The unit test should now appear in **Test Explorer**. Right-click it and select **Run Selected Tests**.



Note that while the test process is running, the progress bar in **Test Explorer** shows an indeterminate green indicator.



13. However, once the test fails, the bar turns red. This provides a quick and easy way to see the status of your tests. You can also see the status of each individual test based on the icon to its left. Click the test result to see the details about why it failed. If you click the **CalculatorTests.AddSimple** link at the bottom, it will bring you to the method.

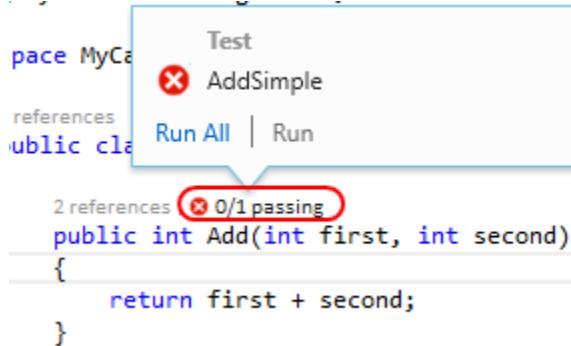


The status indicator also appears in the **CodeLens** directly above the method definition for the test in **CalculatorTests.cs**. Note that **CodeLens** is a feature of Visual Studio Ultimate.

```
[TestMethod]
✖ [0 references]
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(0, sum);
}
```

And if you switch to **Calculator.cs**, you'll see the unit testing **CodeLens** that indicates the rate of passing (**0/1 passing** here).

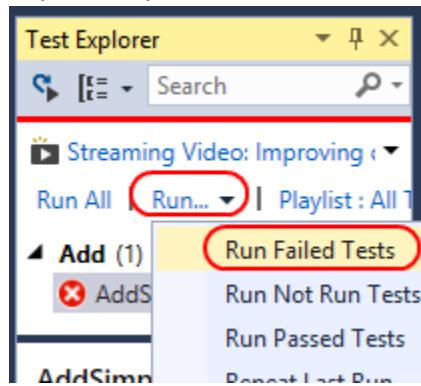
14. Click the **CodeLens** to see the results of each test that exercises this method. If you double-click the **AddSimple** test in the **CodeLens** display, it will bring you directly to the test definition.



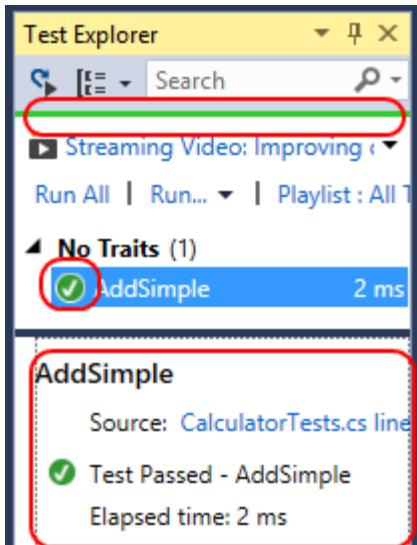
15. In **CalculatorTests.cs**, change the **0** in the **Assert** line to the correct **3**.

C#
Assert.AreEqual(3, sum);

16. In **Test Explorer**, click **Run | Run Failed Tests**. Note that this option only runs the tests that failed in the last pass, which can save time. However, it doesn't run passed tests that may have been impacted by more recent code changes, so be careful when selecting which tests to run.



17. Now that the test passes, the progress bar should provide a solid green. Also, the icon next to the test will turn green. If you click the test, it will provide the basic stats for the test's run.



Also note that the **CodeLens** updates above the test's method definition.

```
[TestMethod]
✓ 0 references
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(3, sum);
}
```

As well as in the **CodeLens** above the library method being tested in **Calculator.cs**.

```
pace MyCa
✓ 2 references 1/1 passing
public class Calculator
{
    public int Add(int first, int second)
    {
        return first + second;
    }
}
```

Task 3: Creating a new feature using test-driven development

In this task, you'll create a new feature in the calculator library using the philosophy known as "test-driven development" (TDD). Simply put, this approach encourages that the tests be written before new code is developed, such that the initial tests fail and the new code is not complete until all the tests pass. Some developers find this paradigm to produce great results, while others prefer to write tests during or after a new feature is implemented. It's really up to you and your organization because Visual Studio provides the flexibility for any of these approaches.

1. Add the following method to **CalculatorTests.cs**. It is a simple test that exercises the **Divide** method of the library.

```
C#
[TestMethod]
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

2. However, since the **Divide** method has not yet been built, the test cannot be run. However, you can feel confident that this is how it's supposed to work, so now you can focus on implementation.

```
[TestMethod]
0 references
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

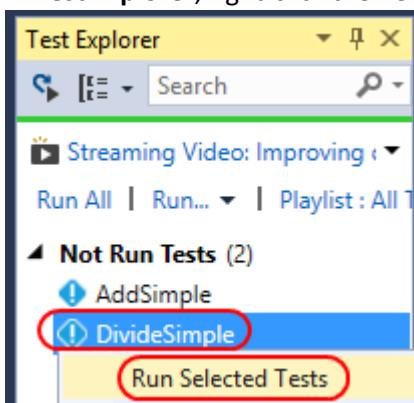
Strictly speaking, the next step should be to implement only the method shell with no functionality. This will allow you to build and run the test, which will fail. However, you can skip ahead here by adding the complete method since it's only one line.

3. Add the **Divide** method to **Calculator.cs**.

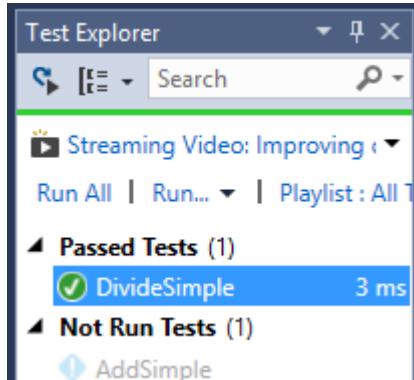
```
C#
public int Divide(int dividend, int divisor)
{
    return dividend / divisor;
}
```

4. From the main menu, select **Build | Build Solution**.

5. In **Test Explorer**, right-click the new **DivideSimple** method and select **Run Selected Tests**.



The test should complete successfully.

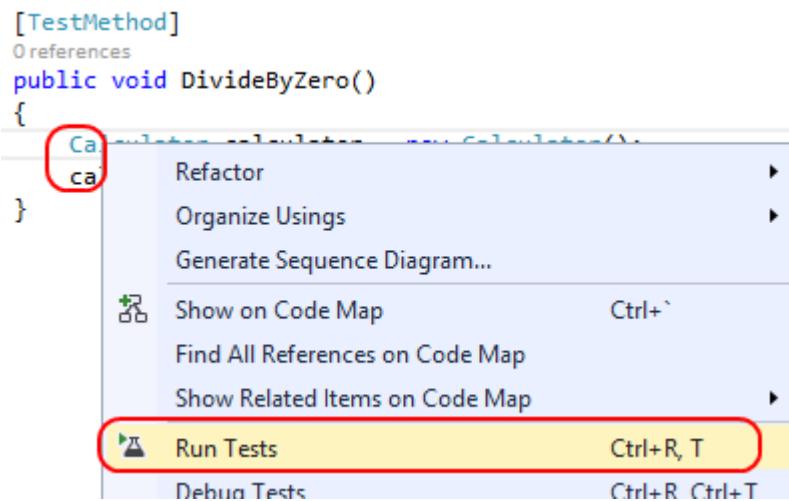


However, there is one edge case to be aware of, which is the case where the library is asked to divide by zero. Ordinarily you would want to test the inputs to the method and throw exceptions as needed, but since the underlying framework will throw the appropriate **DivideByZeroException**, you can take this easy shortcut.

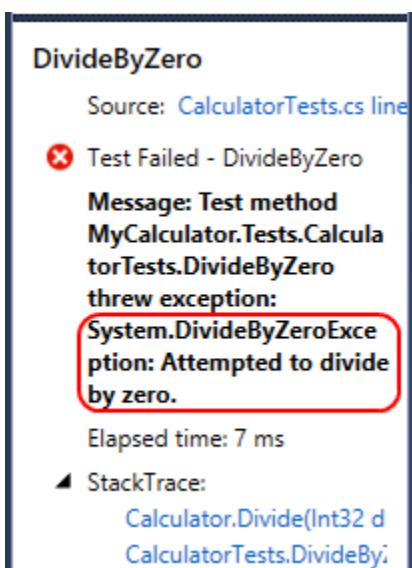
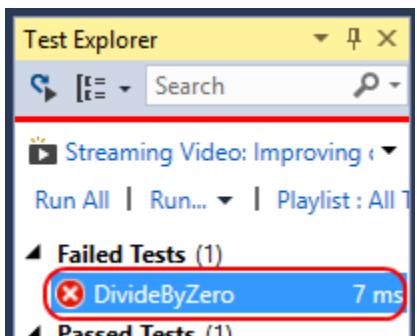
6. Add the following method to **CalculatorTests.cs**. It attempts to divide 10 by 0.

```
C#
[TestMethod]
public void DivideByZero()
{
    Calculator calculator = new Calculator();
    calculator.Divide(10, 0);
}
```

7. Right-click within the body of the test method and select **Run Tests**. This is a convenient way to run just this test.



8. The test will run and fail, but that's expected. If you select the failed test in **Test Explorer**, you'll see that the **DivideByZeroException** was thrown, which is by design.

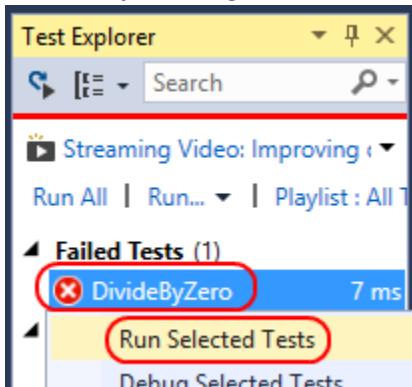


However, this is expected behavior, so you can attribute the test method with the **ExpectedException** attribute.

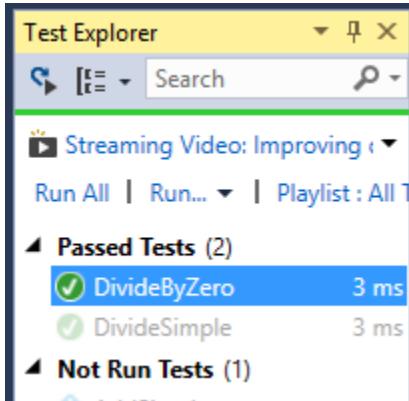
9. Add the following attribute above the definition of **DivideByZero**. Note that it takes the type of exception it's expecting to be thrown during the course of the test.

```
C#
[ExpectedException(typeof(DivideByZeroException))]
```

10. In **Test Explorer**, right-click the failed test and select **Run Selected Tests**.



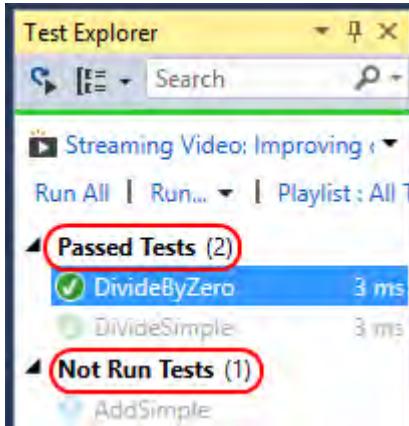
The test should now succeed because it threw the kind of exception it was expecting.



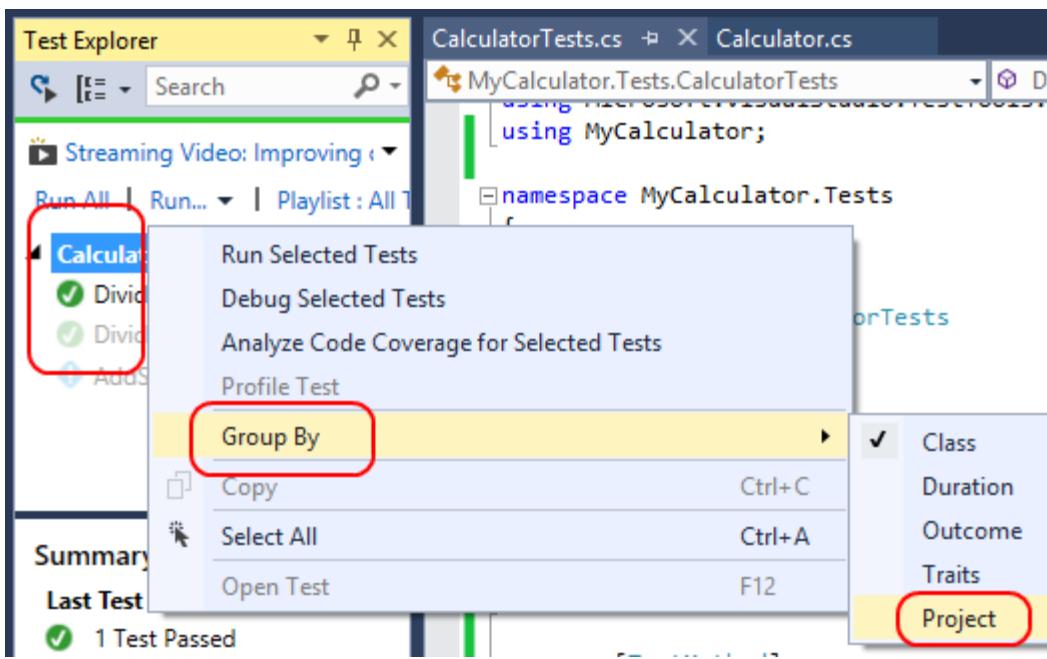
Task 4: Organizing unit tests

In this task, you'll organize the unit tests created so far using different groupings, as well as create custom traits for finer control over organization.

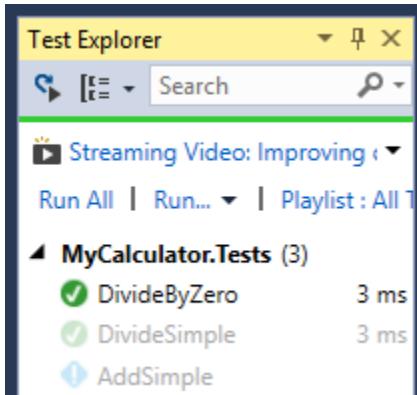
By default, **Test Explorer** organizes tests into three categories: **Passed Tests**, **Failed Tests**, and **Not Run Tests**. This is useful for most scenarios, especially since developers often only care about the tests that are currently failing.



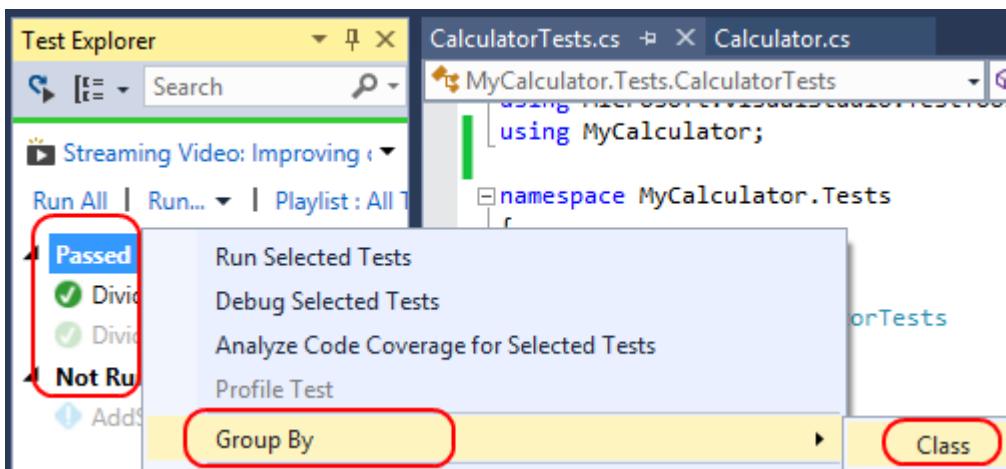
1. However, sometimes it can be useful to group tests by other attributes. Right-click near the tests and select **Group By | Project**.



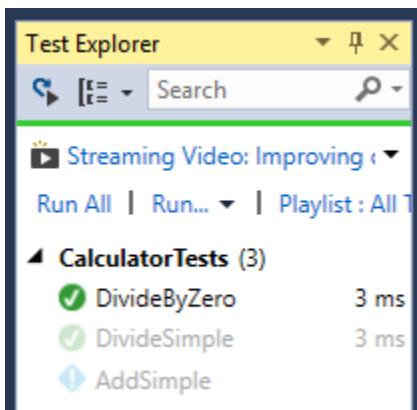
This will organize the tests based on the project they belong to. This can be very useful when navigating huge solutions with many test projects.



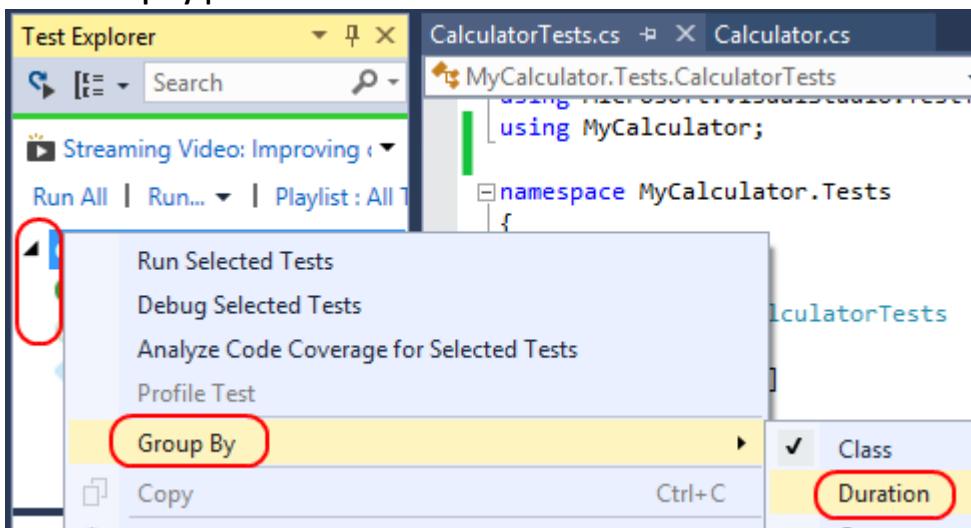
2. Another option is to organize tests by the class they're part of. Right-click near the tests and select **Group By | Class**.



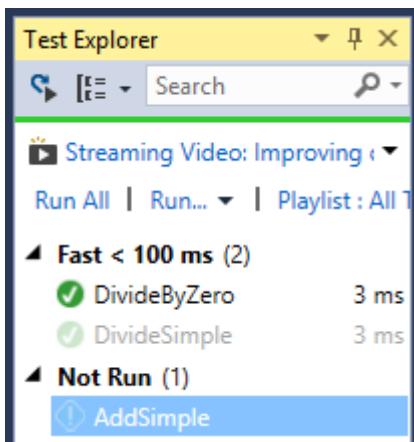
The tests are now grouped by class, which is useful when the classes are cleanly divided across functional boundaries.



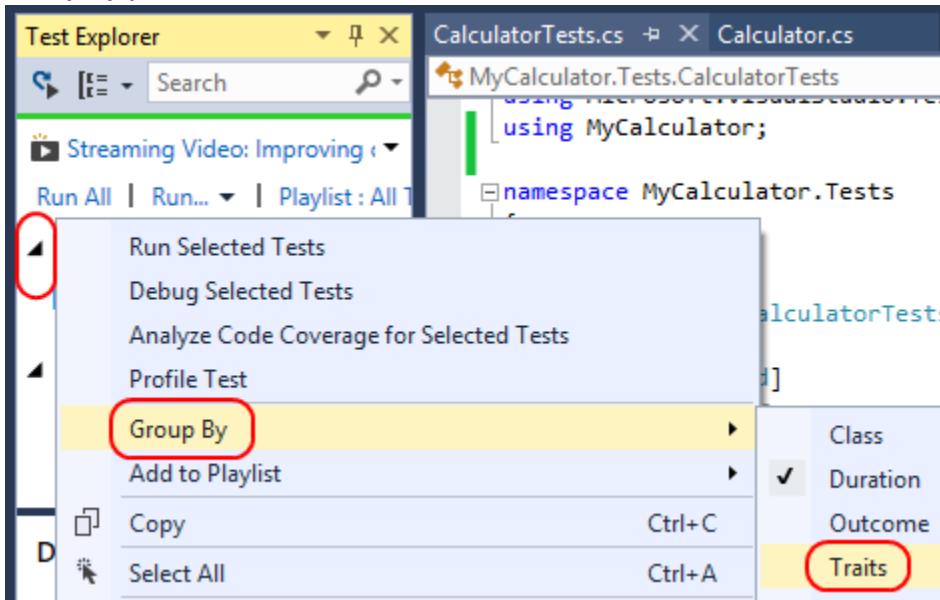
3. Yet another option is to organize tests by how long they take to run. Right-click near the tests and select **Group By | Duration**.



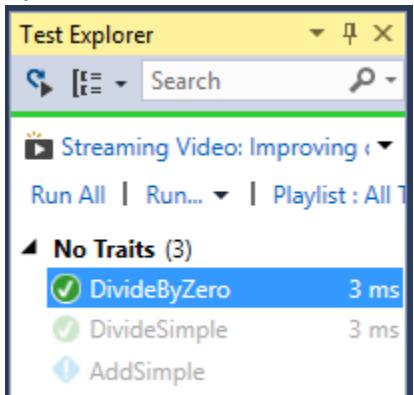
Grouping by duration makes it easy to focus on tests that may indicate poorly performing code.



- Finally, there is another option to organize tests by their **traits**. Right-click near the tests and select **Group By | Traits**.



By default, a test method doesn't have any traits.

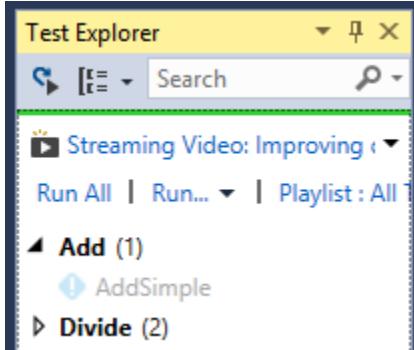


5. However, traits are easy to add using an attribute. Add the following code above the **AddSimple** method. It indicates that this test has the **Add** trait. Note that you may add multiple traits per test, which is useful if you want to tag a test with its functional purpose, development team, performance relevance, etc.

C#

```
[TestCategory("Add")]
```

6. Add **TestCategory** attributes to each of the divide tests, but use the category **Divide**.
7. From the main menu, select **Build | Build Solution**.

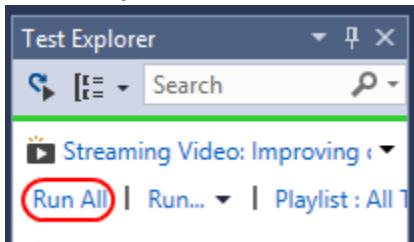


8. Another useful attribute for organizing tests is the **Ignore** attribute. This simply tells the test engine to skip this test when running. Add the following code above the **AddSimple** method.

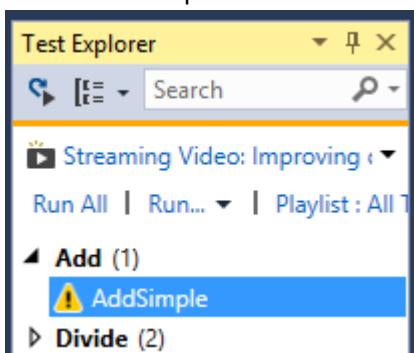
C#

```
[Ignore]
```

9. In **Test Explorer**, click **Run All** to build and run all tests.



10. Note that **AddSimple** has a yellow exclamation icon now, which indicates that it did not run as part of the last test pass.



There are a few more test attributes that aid in the organization and tracking of unit tests.

- **Description** allows you to specify a description for the test.
- **Owner** specifies the owner of the test.
- **HostType** allows you to specify the type of host the test can run on.
- **Priority** specifies the priority at which this test should run.
- **Timeout** specifies how long the test may run until it times out.
- **WorkItem** allows you to specify the work item IDs the test is associated with.
- **CssProjectStructure** represents the node in the team project hierarchy to which this test corresponds.
- **CssIteration** represents the project iteration to which this test corresponds.

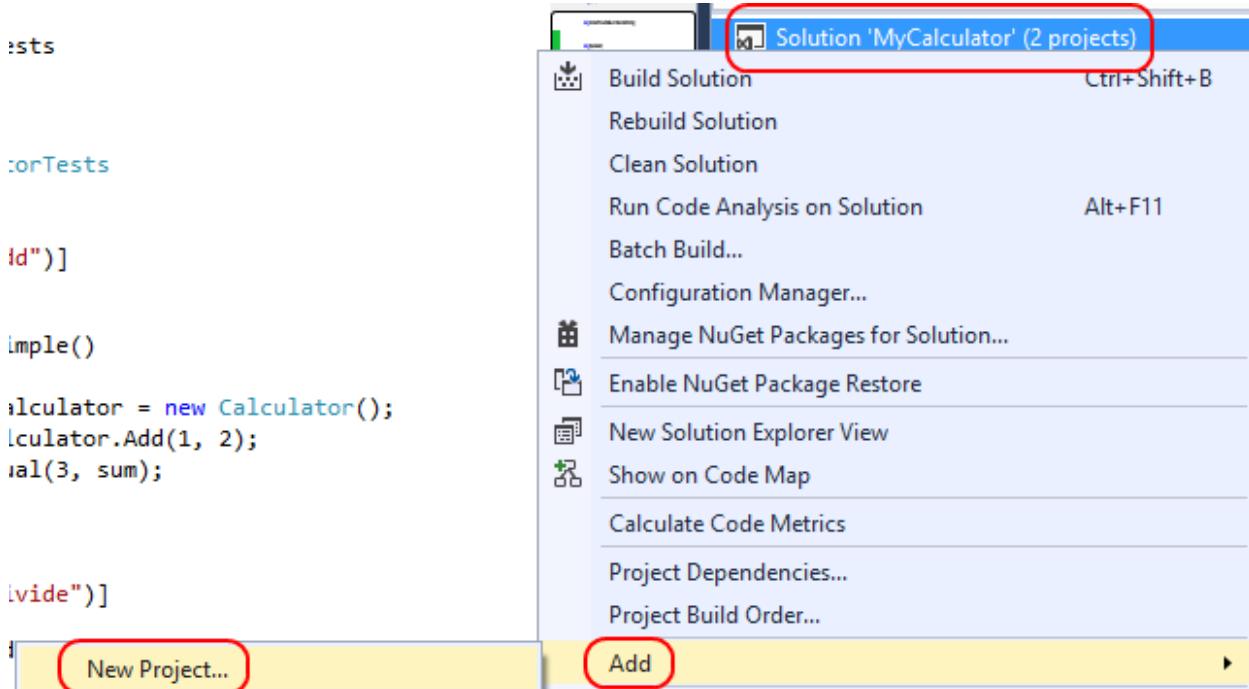
Exercise 2: Unit testing user applications

In this exercise, you'll go through the process of working with unit tests on applications with user interfaces. While it has been historically difficult to build robust tests for applications like these, proper design considerations can drastically improve the testability of virtually any application.

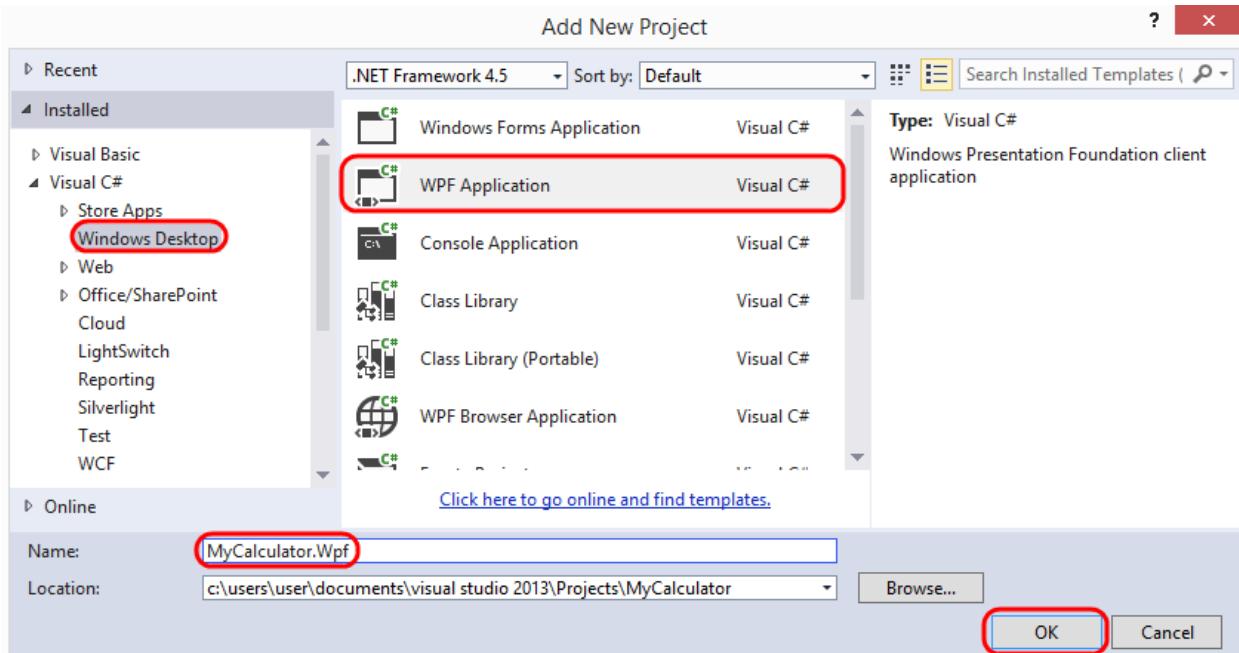
Task 1: Creating a user interface

In this task, you'll create a basic user interface for your library. The goal will be to deploy a user interface as quickly as possible, and no special consideration will be given for testability.

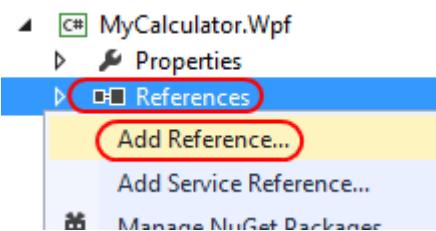
1. In **Solution Explorer**, right-click the solution node and select **Add | New Project....**



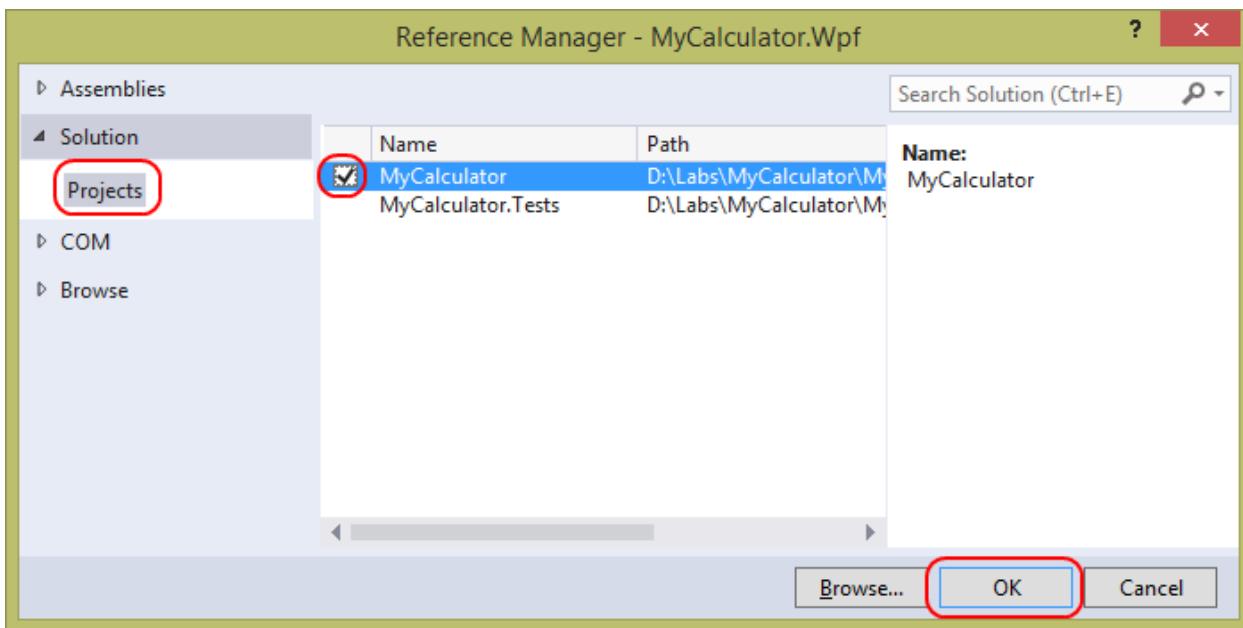
2. Select **Visual C# | Windows Desktop** as the category and **WPF Application** as the template. Type "**MyCalculator.Wpf**" as the Name and click **OK**.



3. The first thing you'll need to do is to add a reference to the **MyCalculator** library. In **Solution Explorer**, right-click the **References** node under **MyCalculator.Wpf** and select **Add Reference...**.



4. Select **Projects** from the left pane and check **MyCalculator** from the assembly list. Click **OK** to add the reference.

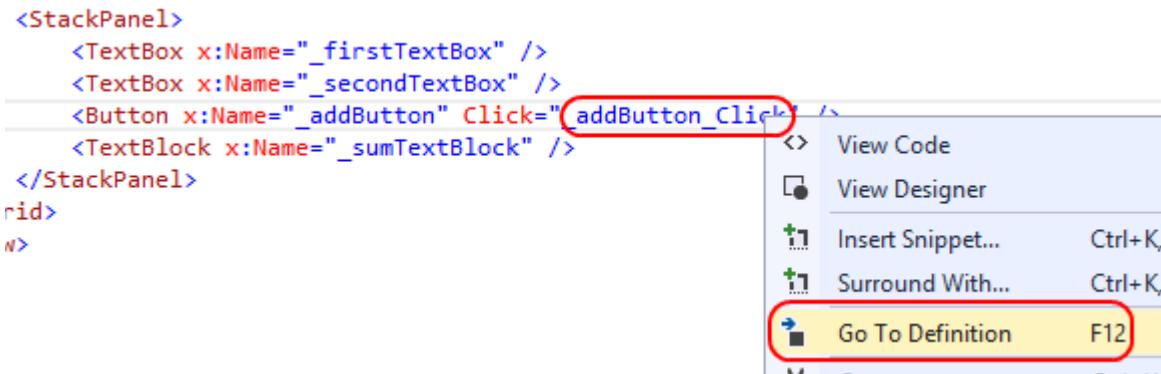


5. The user interface for this calculator application will be very simple with two text boxes, a button, and a text block for the result. Add the following XAML inside the **Grid** node of **MainWindow.xaml**.

XAML

```
<StackPanel>
    <TextBox x:Name="_firstTextBox" />
    <TextBox x:Name="_secondTextBox" />
    <Button x:Name="_ addButton" Content="Add" Click="_ addButton_Click" />
    <TextBlock x:Name="_sumTextBlock" />
</StackPanel>
```

6. The application will access the **MyCalculator** library functionality from the code-behind, so right-click **_ addButton_Click** and select **Go To Definition**.



7. Inside **_ addButton_Click**, add the following code. It parses the values from the text blocks and uses the calculator library to add them. The result is then placed into the text block.

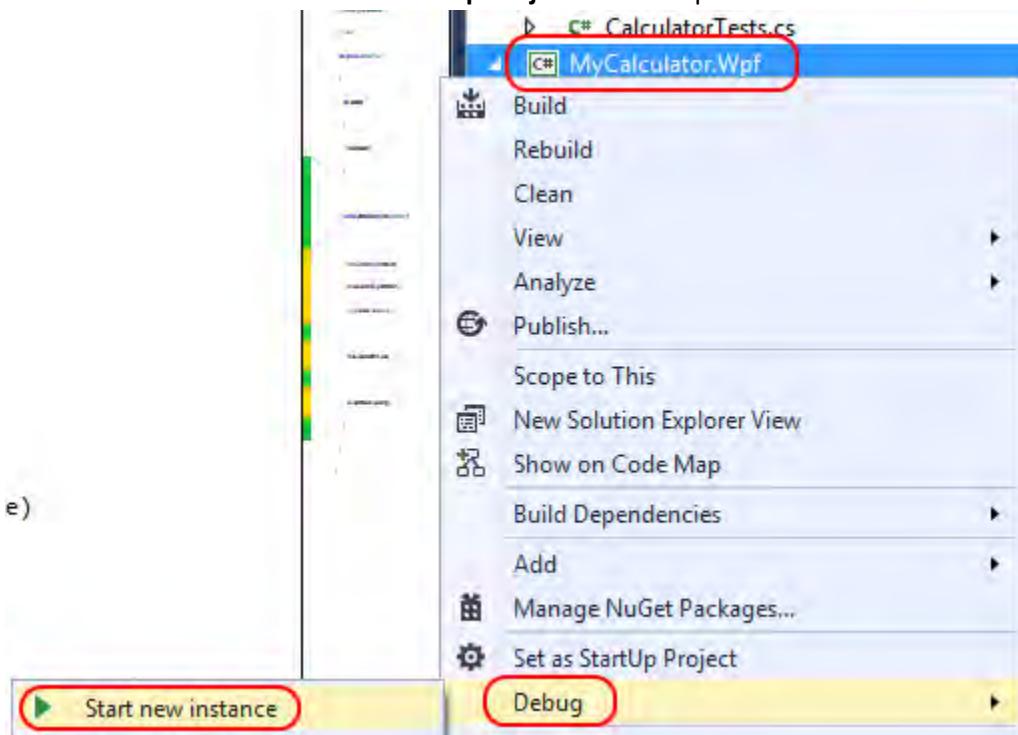
```
C#
int first = int.Parse(this._firstTextBox.Text);
```

```
int second = int.Parse(this._secondTextBox.Text);
Calculator calculator = new Calculator();

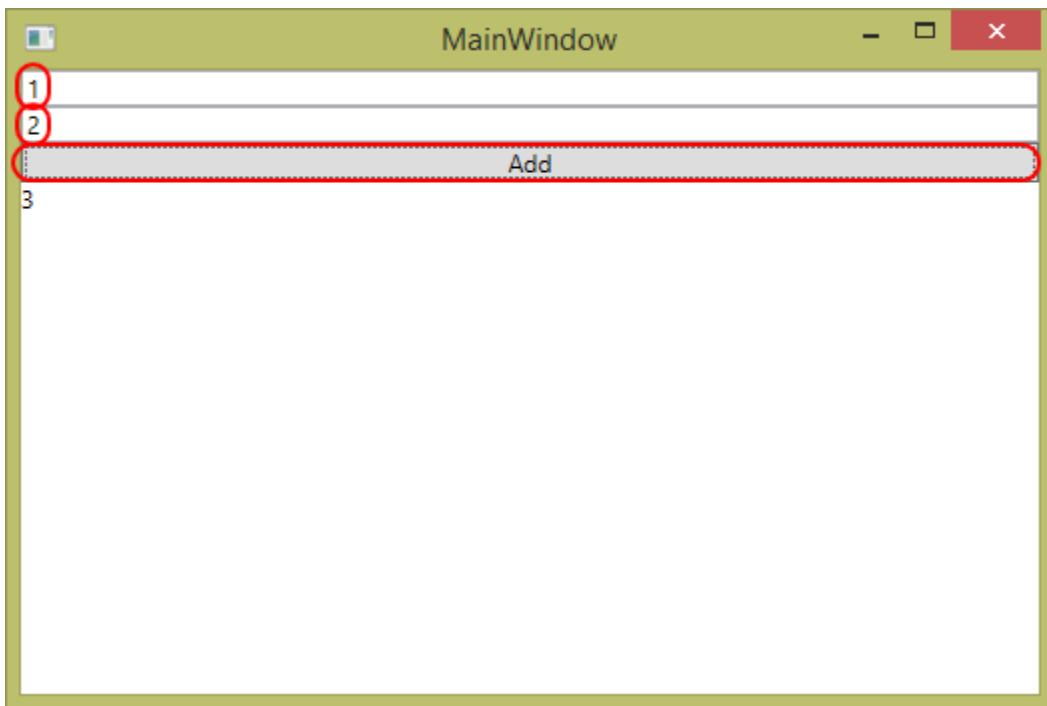
int sum = calculator.Add(first, second);

this._sumTextBlock.Text = sum.ToString();
```

8. In **Solution Explorer**, right-click **MyCalculator.Wpf** and select **Debug | Start new instance**. Note that you're only going to run this application once in this lab. If you were running it multiple times, it would be easier to select **Set as StartUp Project** and then press **F5** to run it each time.

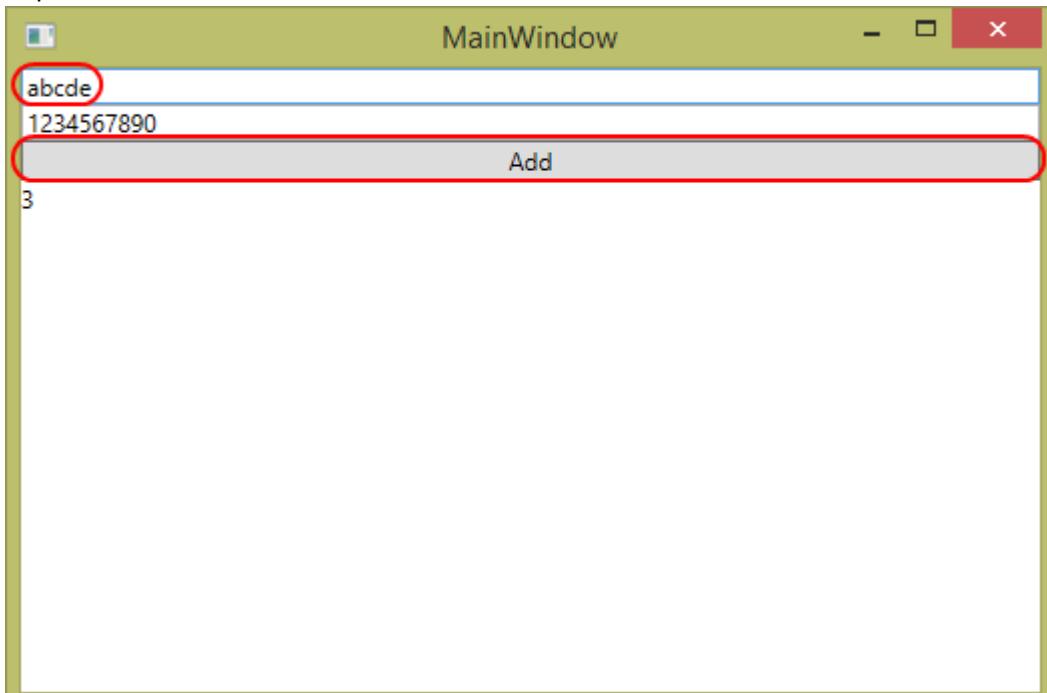


9. The application will build and run. First, use the values **1** and **2** to confirm that **Add** displays **3**. So far so good.

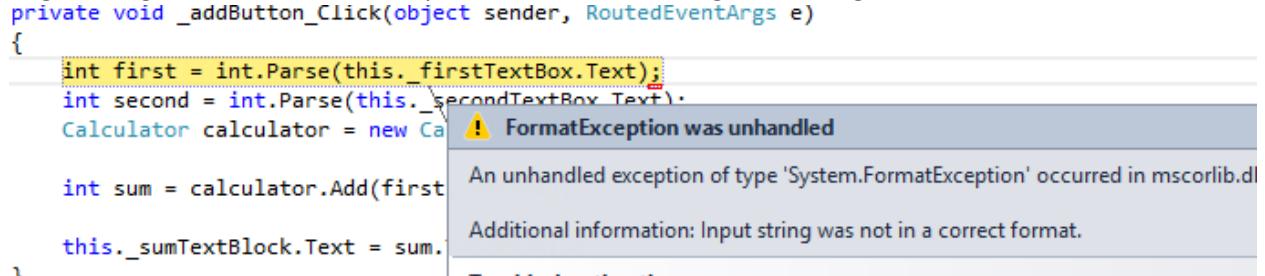


However, while the library supports its functionality fairly well (and you have the unit tests to prove it), there is some untested code that resides in the **MyCalculator.Wpf** project.

10. Replace the value in the first text box with “**abcde**” and click **Add**.



Unfortunately, this results in an exception. In this case, there is a bug in the code-behind. And while you could take the time to fix it, you could never feel high confidence that there aren't other similar bugs lurking under the surface, only to be discovered during actual usage.



A screenshot of Visual Studio showing a debugger exception dialog. The code in the background is:

```
private void _addButton_Click(object sender, RoutedEventArgs e)
{
    int first = int.Parse(this._firstTextBox.Text);
    int second = int.Parse(this._secondTextBox.Text);
    Calculator calculator = new Ca
    int sum = calculator.Add(first, second);
    this._sumTextBlock.Text = sum.ToString();
}
```

The exception dialog has a yellow exclamation icon and the title "FormatException was unhandled". The message says: "An unhandled exception of type 'System.FormatException' occurred in mscorelib.dll". The additional information is: "Additional information: Input string was not in a correct format."

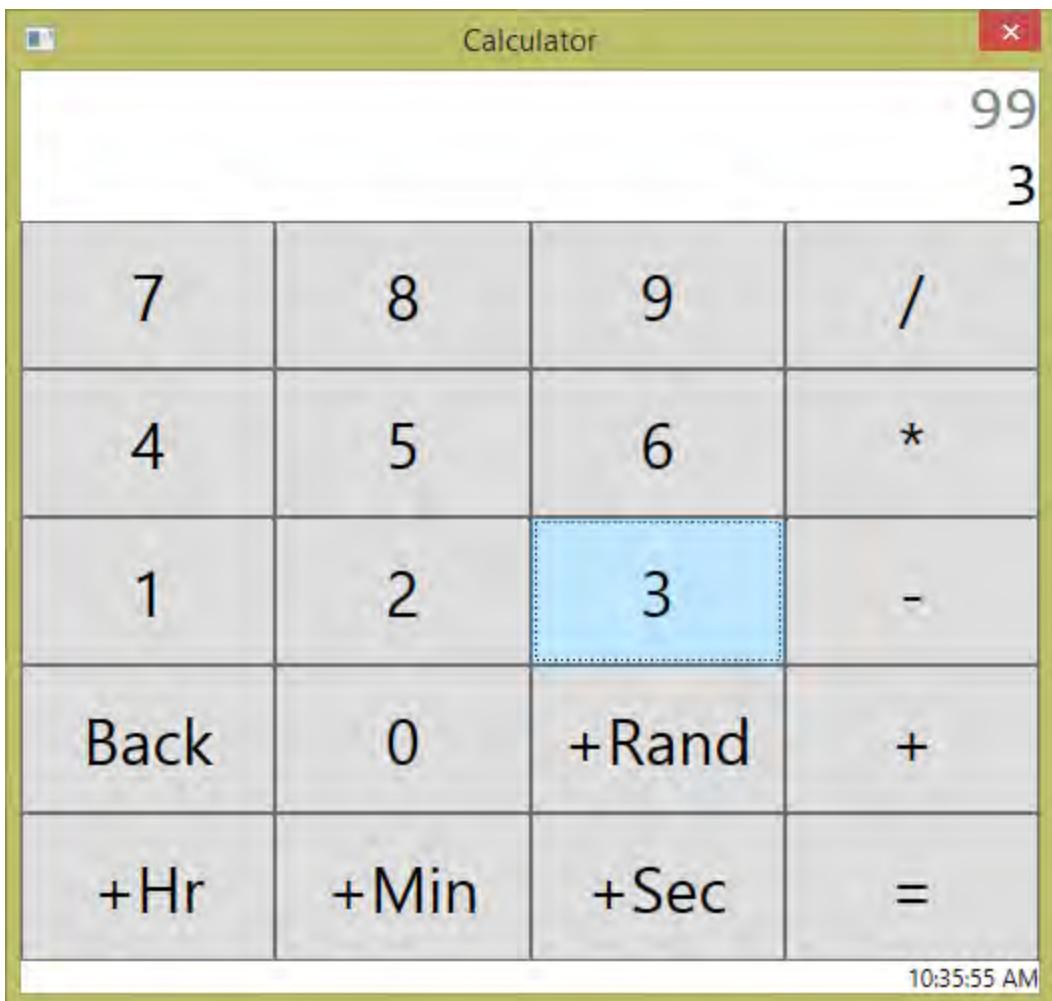
11. From the main menu of Visual Studio, select **Debug | Stop Debugging** to end the debugging session.

Task 2: Designing apps for testability

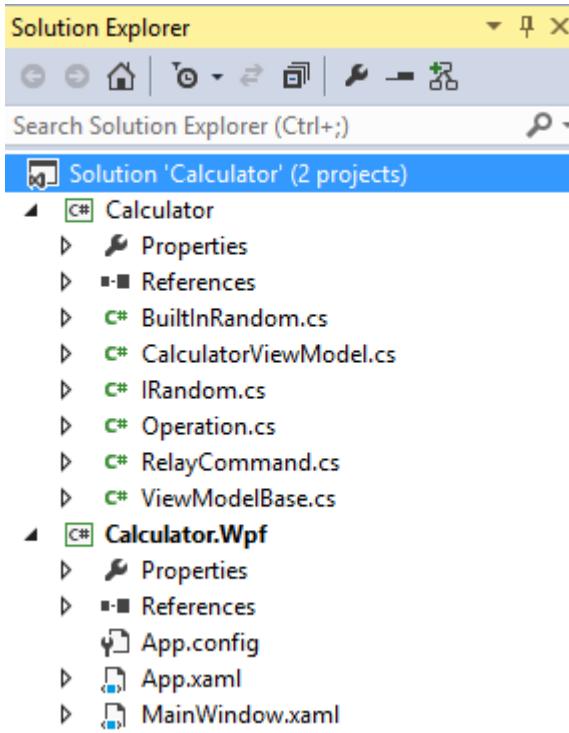
In this task, you'll take a look at a similar calculator application that has made use of a Model-View-ViewModel (MVVM) architecture in order to improve testability. Note that MVVM is outside the scope of this lab, so there will not be a focus on its features or guidance on its usage. However, MVVM represents a class of user applications architectures that promote higher quality software through hardened practices and a focus on test support.

Note: MVVM is an expansive topic that includes benefits that scale well beyond great testability. For more information, please check out Microsoft's official guidance at <http://msdn.microsoft.com/en-us/library/hh848246.aspx>.

1. In Visual Studio, open the solution provided with this lab at **Calculator Solution\Calculator.sln**.
2. Press **F5** to build and run the application. It's a simple calculator that supports integer operations. In addition to the basic four functions, it also allows you to add a random number from 1-100 to the current register, as well as add the current time's hours, minutes, or seconds. Take a few moments to play around with the functionality. Close the application when satisfied.

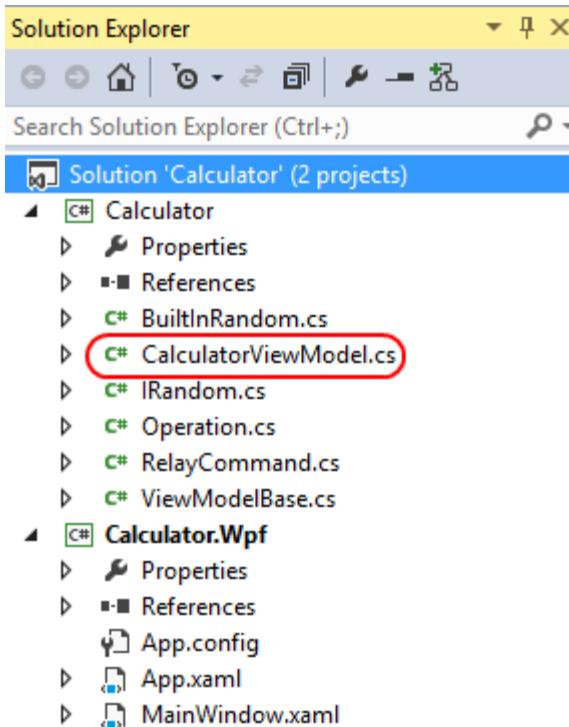


3. In **Solution Explorer**, expand the project nodes to show all top-level files.



This solution has two projects. The **Calculator** project is a class library that contains the logic for the application, and the user interface is contained in **Calculator.Wpf**.

4. Open **CalculatorViewModel.cs**.



In **Calculator**, the main file of interest is the **CalculatorViewModel** class, which is used to perform calculations and communicate with the user interface (from the **Calculator.Wpf** project) via **databinding** and **commanding**.

5. Locate the section where the **ICommand** properties are defined.

```
3 references
public ICommand KeyCommand { get; private set; }
6 references
public ICommand AddCommand { get; private set; }
2 references
public ICommand SubtractCommand { get; private set; }
2 references
public ICommand MultiplyCommand { get; private set; }
2 references
public ICommand DivideCommand { get; private set; }
6 references
public ICommand EquateCommand { get; private set; }
2 references
public ICommand BackCommand { get; private set; }
1 reference
public ICommand AddHoursCommand { get; private set; }
1 reference
public ICommand AddMinutesCommand { get; private set; }
2 references
public ICommand AddSecondsCommand { get; private set; }
2 references
public ICommand AddRandomCommand { get; private set; }
```

These properties are all intended to be wired directly up to the user interface. As a result, there is virtually no code-behind. If you want to test what happens when a user clicks a series of buttons (or performs a series of other input actions not leveraged here), you can use the **Execute** method each **ICommand** exposes. Note that this all happens with no user interface present, but it doesn't matter since the **CalculatorViewModel** is abstracted away from the classes using it.

Another important pattern to rely on where possible is known as **Dependency Injection**.

6. Locate the private **IRandom** property, which is just above the constructor.

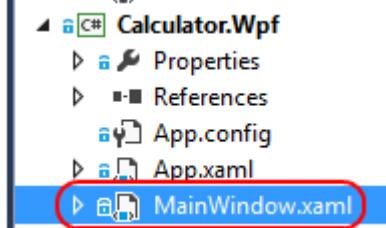
```
private IRandom _random;

3 references
public CalculatorViewModel(IRandom random)
{
    // Initializing values and resources.
    this._random = random;
    ...
```

The **IRandom** interface provides an abstraction layer for the library to generate random numbers. While the library itself provides an implementation that uses **System.Random** in **BuiltInRandom**, you could pass in any **IRandom** you want. As a result, you have much more flexibility when it comes to controlling the behavior of the class for isolation and testing purposes.

In **Calculator.Wpf**, virtually all of the functionality is wired up in **MainWindow.xaml**, which binds its user interface (data and buttons) to an instance of **CalculatorViewModel**, which is set in **MainWindow.xaml**.

7. From **Solution Explorer**, open **MainWindow.xaml**.



8. Locate the two **TextBlocks** near the top of the file. They display the stored and current registers of the calculator. However, their values are databound to the properties exposed by the **CalculatorViewModel**, so there's no additional code between the XAML and ViewModel.

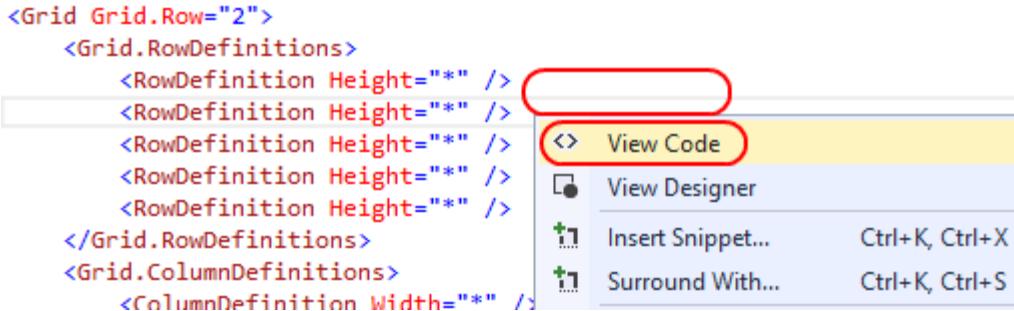
```
<TextBlock Grid.Row="0" FontFamily="Consolas" FontSize="32" Text="{Binding StoredValue}" For  
<TextBlock Grid.Row="1" FontFamily="Consolas" FontSize="32" Text="{Binding CurrentValue}" Te
```

9. Locate the long list of **Button** elements.

```
<Button Content="7" Grid.Row="0" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="7"  
<Button Content="8" Grid.Row="0" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="8"  
<Button Content="9" Grid.Row="0" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="9"  
<Button Content="/" Grid.Row="0" Grid.Column="3" FontSize="32" Command="{Binding DivideCommand}" />  
<Button Content="4" Grid.Row="1" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="4"  
<Button Content="5" Grid.Row="1" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="5"  
<Button Content="6" Grid.Row="1" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="6"  
<Button Content="*" Grid.Row="1" Grid.Column="3" FontSize="32" Command="{Binding MultiplyCommand}" />  
<Button Content="1" Grid.Row="2" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="1"  
<Button Content="2" Grid.Row="2" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="2"  
<Button Content="3" Grid.Row="2" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="3"  
<Button Content="-" Grid.Row="2" Grid.Column="3" FontSize="32" Command="{Binding SubtractCommand}" />  
<Button Content="Back" Grid.Row="3" Grid.Column="0" FontSize="32" Command="{Binding BackCommand}" />  
<Button Content="0" Grid.Row="3" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="0"  
<Button Content="+Rand" Grid.Row="3" Grid.Column="2" FontSize="32" Command="{Binding AddRandomCommand}" />  
<Button Content="+" Grid.Row="3" Grid.Column="3" FontSize="32" Command="{Binding AddCommand}" />  
<Button Content="+Hr" Grid.Row="4" Grid.Column="0" FontSize="32" Command="{Binding AddHoursCommand}" />  
<Button Content="+Min" Grid.Row="4" Grid.Column="1" FontSize="32" Command="{Binding AddMinutesCommand}" />  
<Button Content="+Sec" Grid.Row="4" Grid.Column="2" FontSize="32" Command="{Binding AddSecondsCommand}" />  
<Button Content="=" Grid.Row="4" Grid.Column="3" FontSize="32" Command="{Binding EquateCommand}" />
```

Note how all these buttons have **Commands** set instead of **Click** handlers. These connect directly to the ViewModel as discussed earlier, so there's no additional code to worry about between the XAML and the library you can test directly. Another item to note is the **CommandParameter** property set on the number buttons. These parameters allow reuse of the same command for similar behavior. You could also provide a keyboard hook that uses these same commands to send keys pressed by the user, and almost the entire code path would be the same as used by these buttons.

10. Right-click inside the editor and select **View Code** to navigate to the code-behind file, **MainWindow.xaml.cs**.

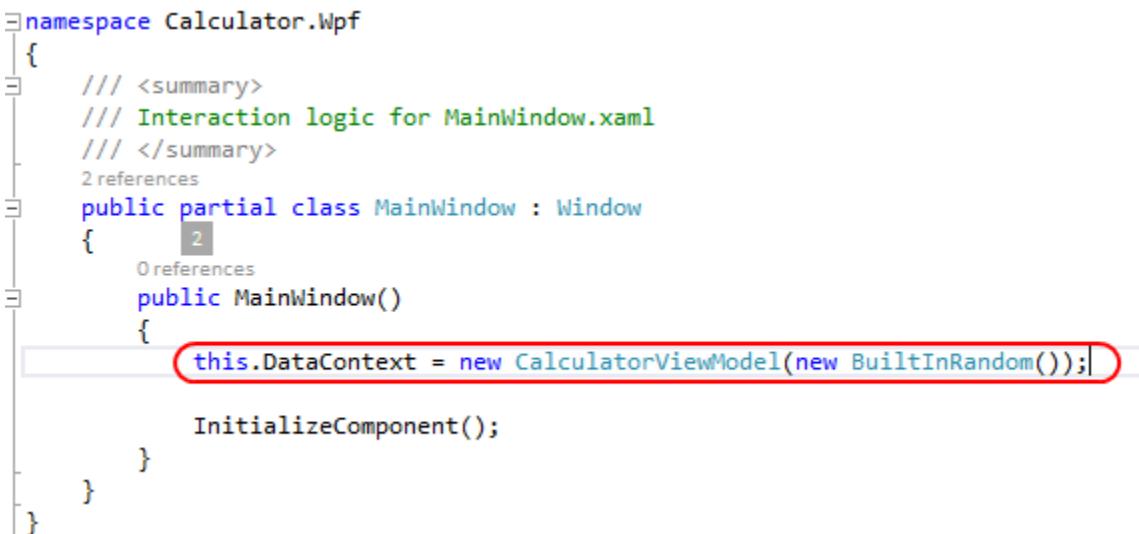


```

<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>

```

11. There is only one line of code that's been added to this entire project, and it simply sets the **DataContext**. After that, everything is handled between the XAML layer and the ViewModel.



```

namespace Calculator.Wpf
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            this.DataContext = new CalculatorViewModel(new BuiltInRandom());
            InitializeComponent();
        }
    }
}

```

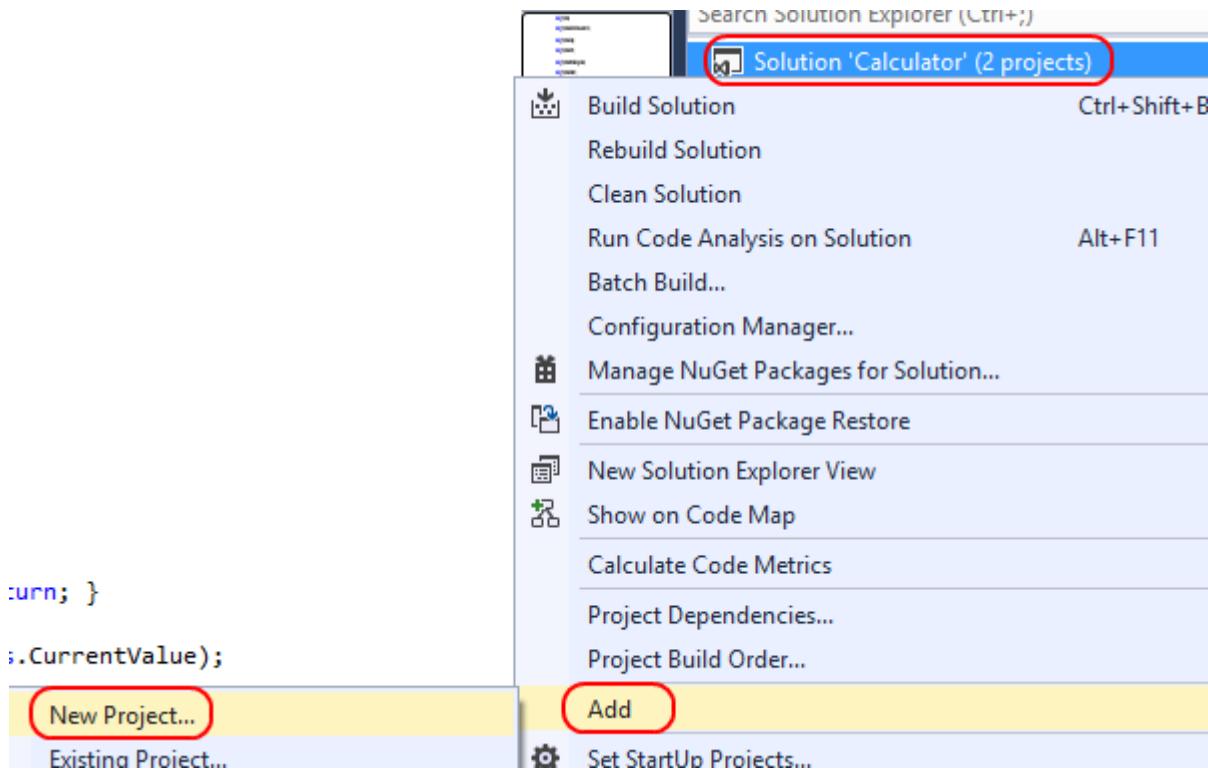
Feel free to take a few minutes to explore the codebase. Keep in mind that the details of the implementation aren't really important in this lab since the focus is on unit testing. The key message is that because this application was designed using MVVM, virtually everything that needs to be tested can be tested in the **Calculator** library, including user behavior such as button clicks.

Note: Another way to test user interfaces is called **Coded UI Tests**. In a coded UI test, the test machine runs automation that invokes mouse and keyboard actions to test the application. These tests can be used alongside unit tests for greater coverage. For more information on coded UI tests, please visit <http://msdn.microsoft.com/en-us/library/dd286726.aspx>.

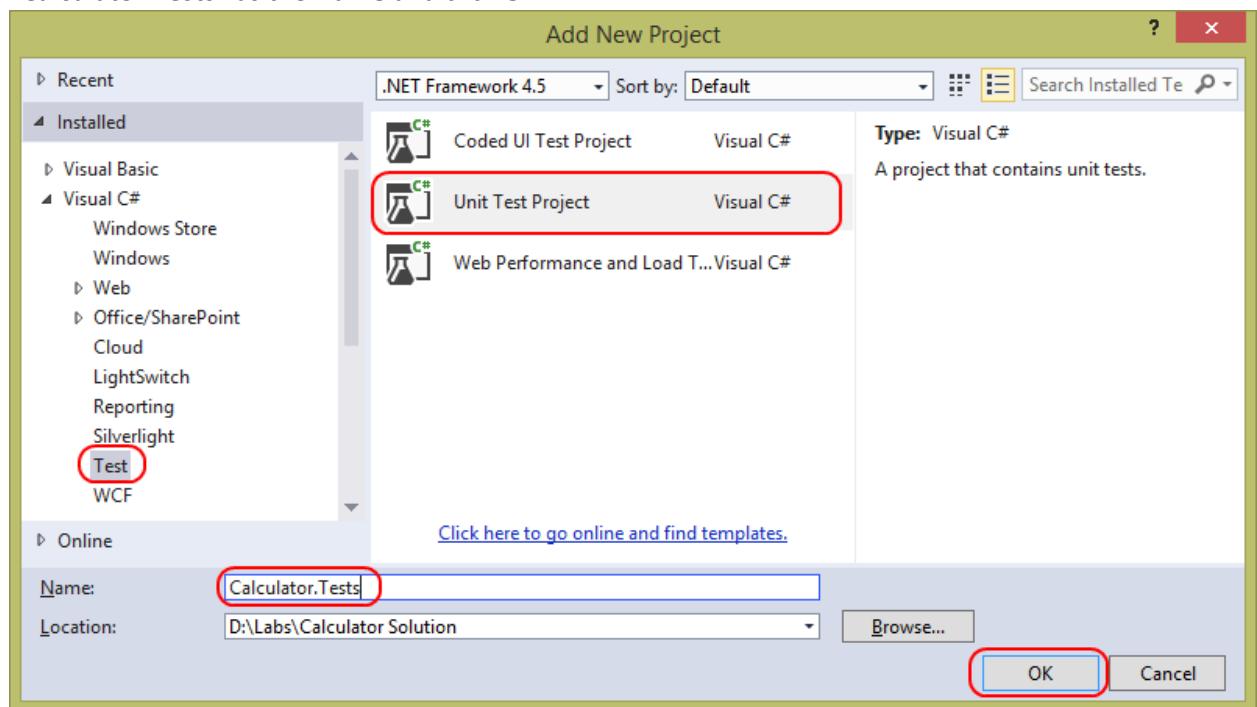
Task 3: Creating tests for the MVVM application

In this task, you'll add a test project and create some more advanced tests for the MVVM application.

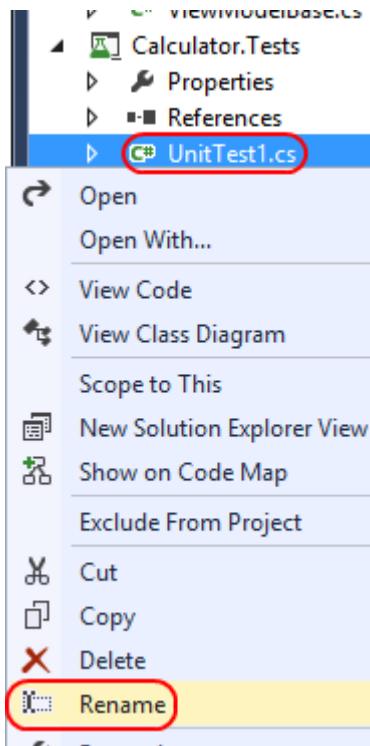
1. In **Solution Explorer**, right-click the solution node and select **Add | New Project....**



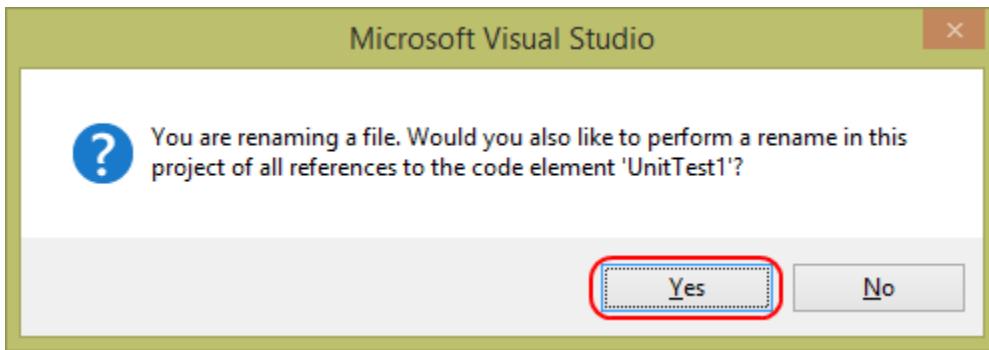
2. Select **Visual C# | Test** as the category and **Unit Test Project** as the template. Type “Calculator.Tests” as the **Name** and click **OK**.



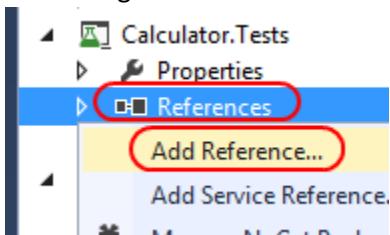
3. Right-click **UnitTest1.cs** and select **Rename**.



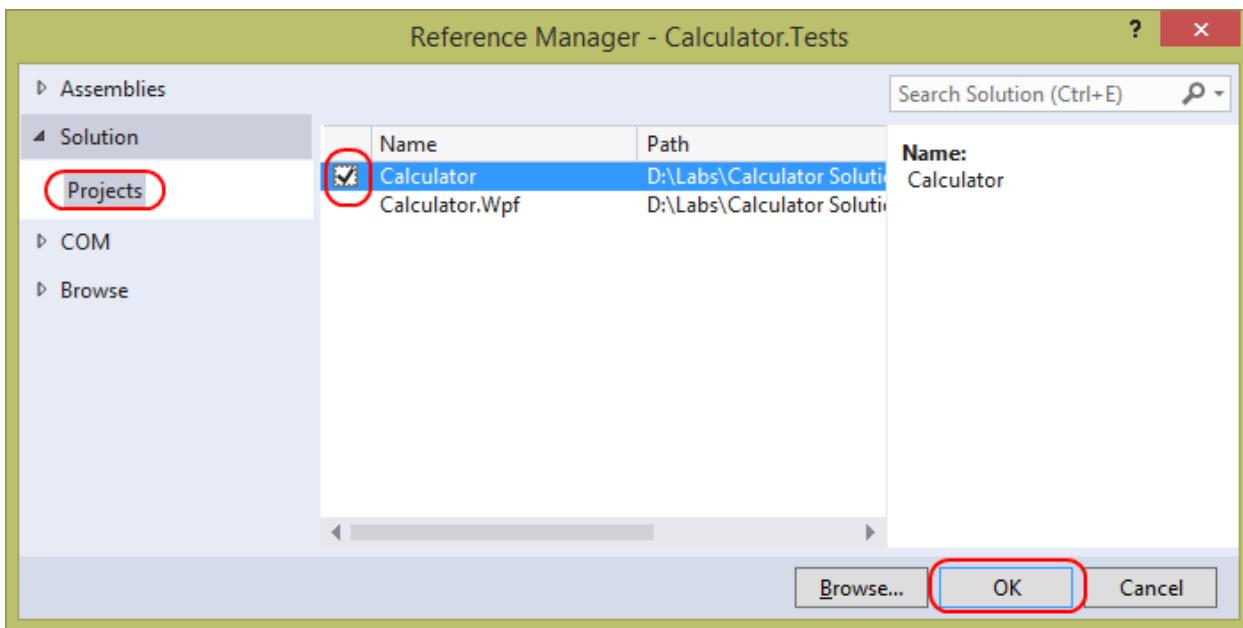
4. Rename the file to “**MathTests.cs**” and click **Enter**. When asked to rename the class itself, click **Yes**.



5. As with the earlier unit test project, you’ll need to add a reference to the calculator library being tested. Right-click the **References** node under **Calculator.Tests** and select **Add Reference....**



6. Select **Projects** from the left pane and check the **Calculator** assembly. Click **OK** to add a reference to it.



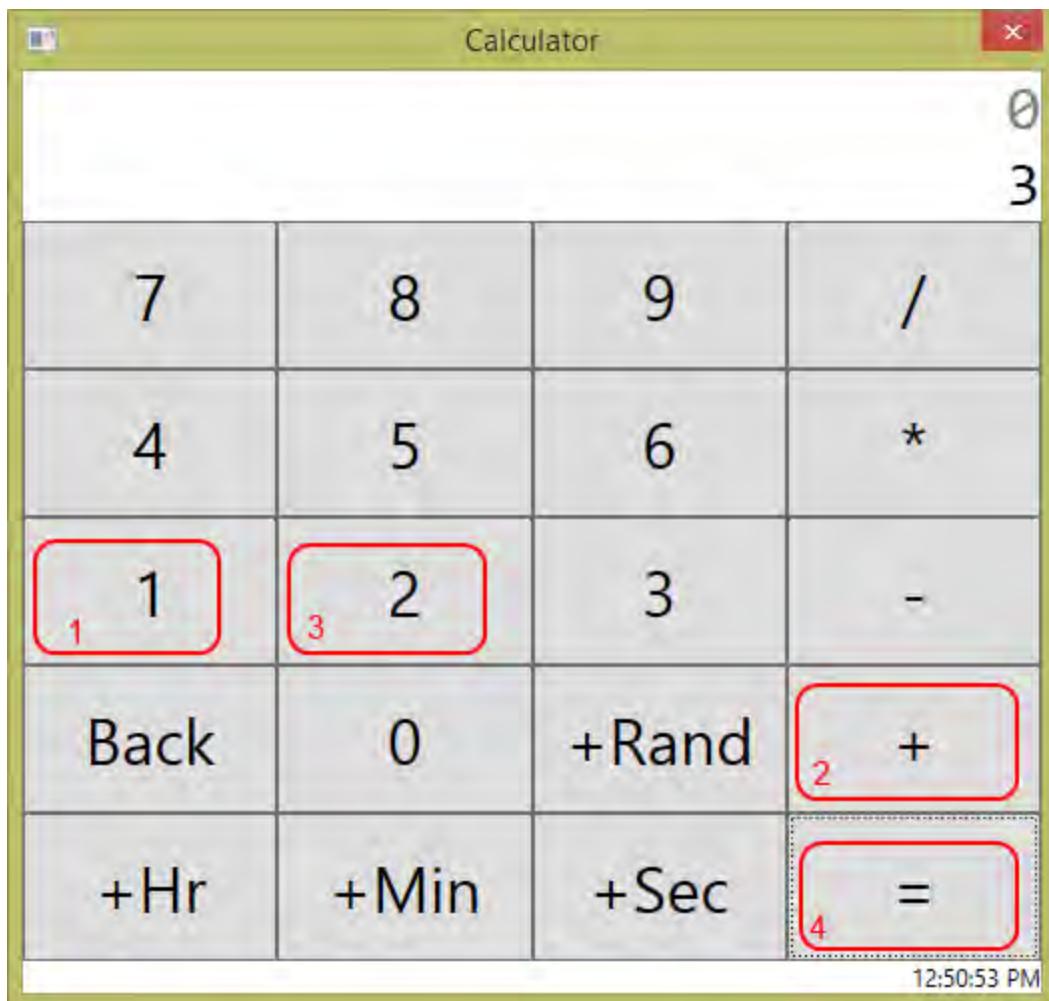
7. Rename the default **TestMethod1** to **AddSimple**.

```
namespace Calculator.Tests
{
    [TestClass]
    public class MathTests
    {
        [TestMethod]
        public void AddSimple()
        {
        }
    }
}
```

Now that everything's set up to start writing the test for **Add** functionality, it's time to write the test itself. As a developer, you'll typically write unit tests for your own code around the same time you write the code itself. However, in this scenario the codebase is inherited without any tests, so you should take a moment to figure out how to interact with the library.

This application uses a fairly strict MVVM approach, so there is no business logic in the WPF codebase. As a result, every button click invokes a command. If you wanted to write a test that closely emulated the way users would interact with the application, you could simply execute those commands as though the unit test were pushing buttons.

8. Press **F5** to run the application.
9. Perform the calculation **1 + 2 =**, which should produce the result **3** as shown.



Note that there were four button clicks to get the result. Since all these clicks mapped to buttons in the **CalculatorViewModel** class, you can emulate it exactly.

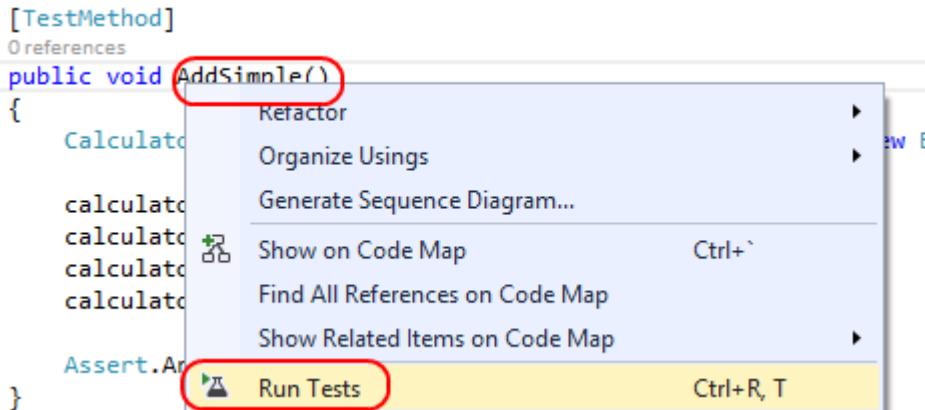
10. Add the following code to the body of **AddSimple**. It'll fail because the sum will actually be **3**.

```
C#
CalculatorViewModel calculator = new CalculatorViewModel(new
BuiltInRandom());

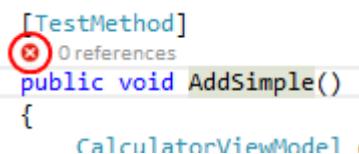
calculator.KeyCommand.Execute("1");
calculator.AddCommand.Execute(null);
calculator.KeyCommand.Execute("2");
calculator.EquateCommand.Execute(null);

Assert.AreEqual(0, calculator.CurrentValue);
```

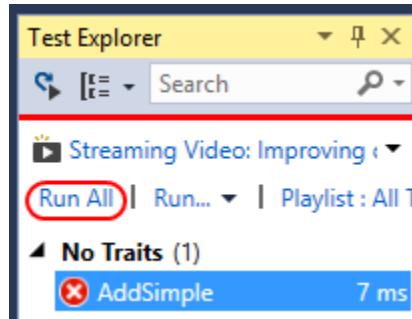
11. Right-click **AddSimple** and select **Run Tests** to run this test.



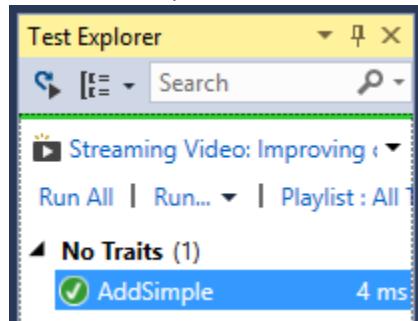
The test fails as expected.



12. Change the **0** in the **Assert** line to **3** to fix the test.
13. In **Test Explorer**, click **Run All**.



The test now passes.



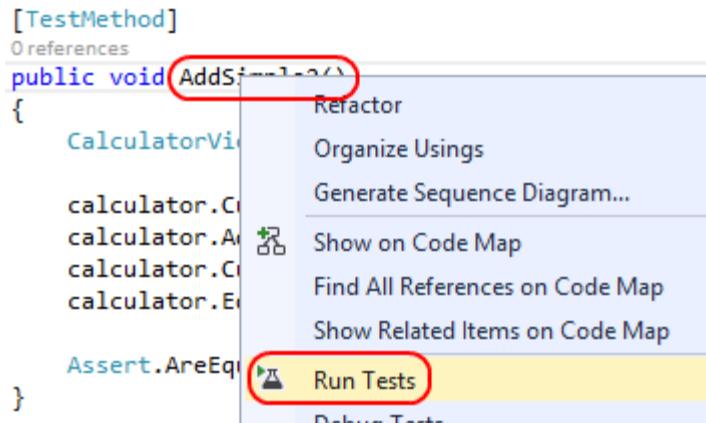
14. Now add an additional addition test by inserting this code into the test class. Note that instead of using the **KeyCommand** steps, the library was designed to allow you to directly set the **CurrentValue** as a shortcut. It's still a good practice to have tests that confirm the key commanding, but it would be unnecessarily complex to have to enter long numbers via emulated key presses.

```
C#
[TestMethod]
public void AddSimple2()
{
    CalculatorViewModel calculator = new CalculatorViewModel(new
    BuiltInRandom());

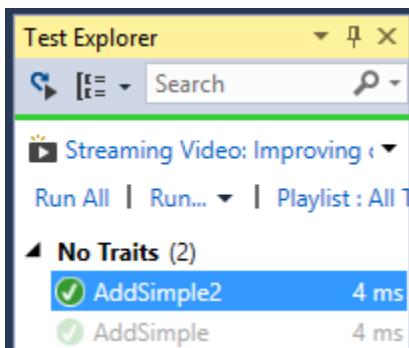
    calculator.CurrentValue = 100;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 200;
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(300, calculator.CurrentValue);
}
```

15. Right-click **AddSimple2** and select **Run Tests**.



The test will succeed.



These two tests have given the **Add** functionality a good bit of exercise, but it would be worthwhile to add in some other functionality for the next test.

16. Add the following test to the class. It performs the four basic functions.

```
C#
[TestMethod]
```

```

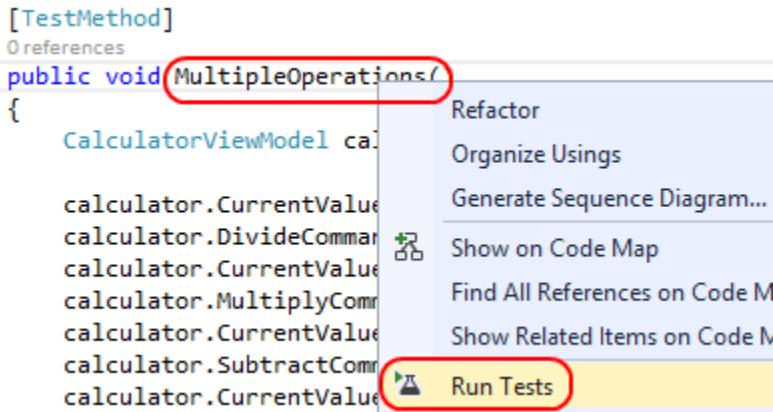
public void MultipleOperations()
{
    CalculatorViewModel calculator = new CalculatorViewModel(new
BuiltInRandom());

    calculator.CurrentValue = 10;
    calculator.DivideCommand.Execute(null);
    calculator.CurrentValue = 2;
    calculator.MultiplyCommand.Execute(null);
    calculator.CurrentValue = 7;
    calculator.SubtractCommand.Execute(null);
    calculator.CurrentValue = 12;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 7;
    calculator.EquateCommand.Execute(null);

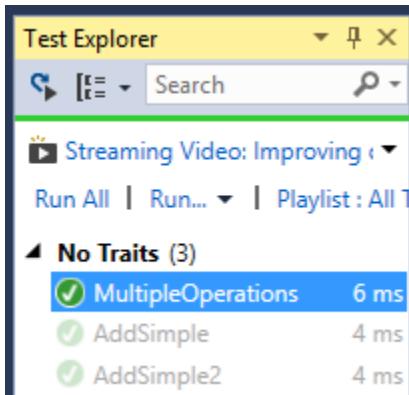
    Assert.AreEqual(30, calculator.CurrentValue);
}

```

17. Right-click **MultipleOperations** and select **Run Tests**.



The test will succeed. However, notice that it likely takes slightly longer due to the extra operations and overhead involved.



Throughout these tests, there has been a bit of redundancy that you can start to factor out. For example, each test is creating a **CalculatorViewModel** using the same **BuiltInRandom** parameter. This isn't necessarily bad form, but since it's so repetitive, you can use a private member and create a new one automatically before the tests.

18. Add the following property near the top of the class.

C#

```
private CalculatorViewModel calculator;
```

19. Add the following initializer method to the class. Note that it's attributed with **TestInitialize**, which indicates that this method should be called prior to each test in the class.

C#

```
[TestInitialize]
public void TestInitialize()
{
    this.calculator = new CalculatorViewModel(new BuiltInRandom());
}
```

In addition to **TestInitialize**, there are five other attributes that allow you to insert a method during the testing lifecycle:

- **TestCleanup** is an attribute for an instance method that runs after each test in that class.
- **ClassInitialize** is an attribute for a static method that runs once prior to any tests in that class.
- **ClassCleanup** is an attribute for a static method that runs once after all the tests in that class have been run.
- **AssemblyInitialize** is an attribute for a static method that runs once prior to any tests in that assembly.
- **AssemblyCleanup** is an attribute for a static method that runs once after all the tests in that assembly have been run.

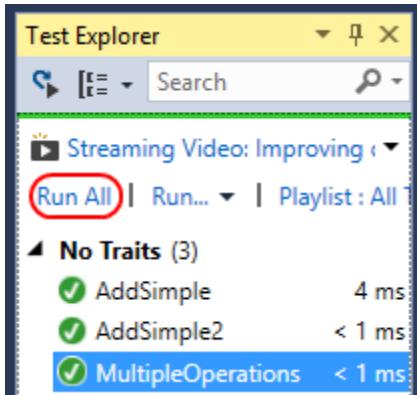
Note: For more details on the proper usage of these attributes, as well as their expected method signatures, please visit <http://msdn.microsoft.com/en-us/library/Microsoft.VisualStudio.TestTools.UnitTesting.aspx>.

20. Remove the **CalculatorViewModel** creation line in each of the existing tests. That variable will now reference the instance member.

C#

```
CalculatorViewModel calculator = new CalculatorViewModel(new
BuiltInRandom());
```

21. In **Test Explorer**, click **Run All** to run all tests. They should all pass as expected.



While MVVM is not the primary focus of this lab, you can see how the up-front investment made in abstracting key components via the MVVM model has paid significant dividends in the form of automated testing. Instead of having to run the UI each time you want to confirm that new code has not introduced a regression bug, you can feel a higher level of confidence that your automated tests provide a great first line of defense.

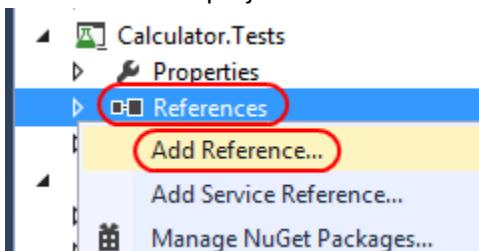
Exercise 3: Advanced unit testing features

In this exercise, you'll learn about some advanced unit testing features in Visual Studio, including data-driven testing, code coverage, and the Fakes framework.

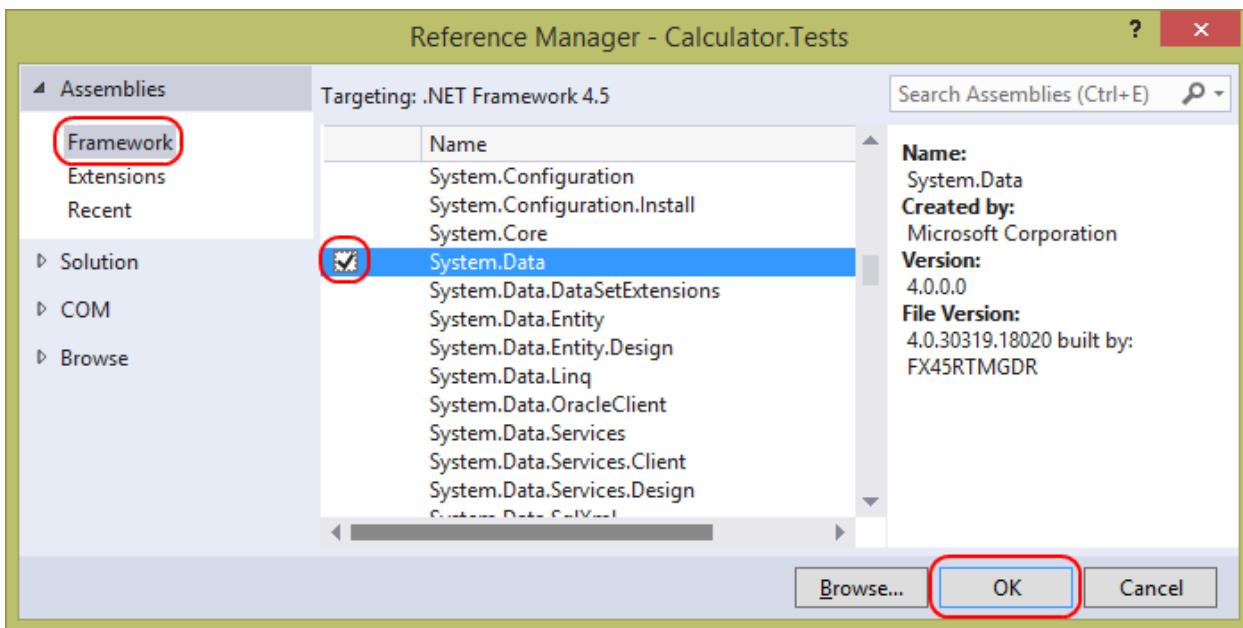
Task 1: Data-driven testing

In this task, you'll create a data-driven test. Data-driven tests provide a great way to perform the same test with different parameters, resulting in broader coverage without duplicated code.

1. Before writing data-driven tests, you'll need to reference an additional assembly that contains some required functionality. In **Solution Explorer**, right-click the **References** node under the **Calculator.Tests** project and select **Add Reference....**



2. From the **Reference Manager** dialog, select the **Assemblies | Framework** category and check the box next to the **System.Data** assembly. Click **OK** to add the reference.



3. Add the following code in **MathTests.cs** to create a data-driven test.

```
C#
[TestMethod]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
    "Tests.xml", "add", DataAccessMethod.Sequential)]
public void AddDataDriven()
{
    int first = int.Parse(this.TestContext.DataRow["first"].ToString());
    int second = int.Parse(this.TestContext.DataRow["second"].ToString());
    int sum = int.Parse(this.TestContext.DataRow["sum"].ToString());

    this.calculator.CurrentValue = first;
    this.calculator.AddCommand.Execute(null);
    this.calculator.CurrentValue = second;
    this.calculator.EquateCommand.Execute(null);
    Assert.AreEqual(sum, this.calculator.CurrentValue);
}
```

Note the use of the **DataSource** attribute, which indicates that this test should be run once per row of data provided in the source.

- The first parameter is the fully qualified name of the data source provider, which is for an XML file here. Alternatively, you can use the appropriate object to support using a database, CSV file, or other source.
- The second parameter is the connection string, which in this case is simply the test-time path to the XML file.
- The third parameter specifies the table in which the data is stored (or in the case of your XML file, will be the name of the elements directly under the root node).

- The final parameter indicates that the tests should be run using the data in the order it was found (the alternative is to randomize).

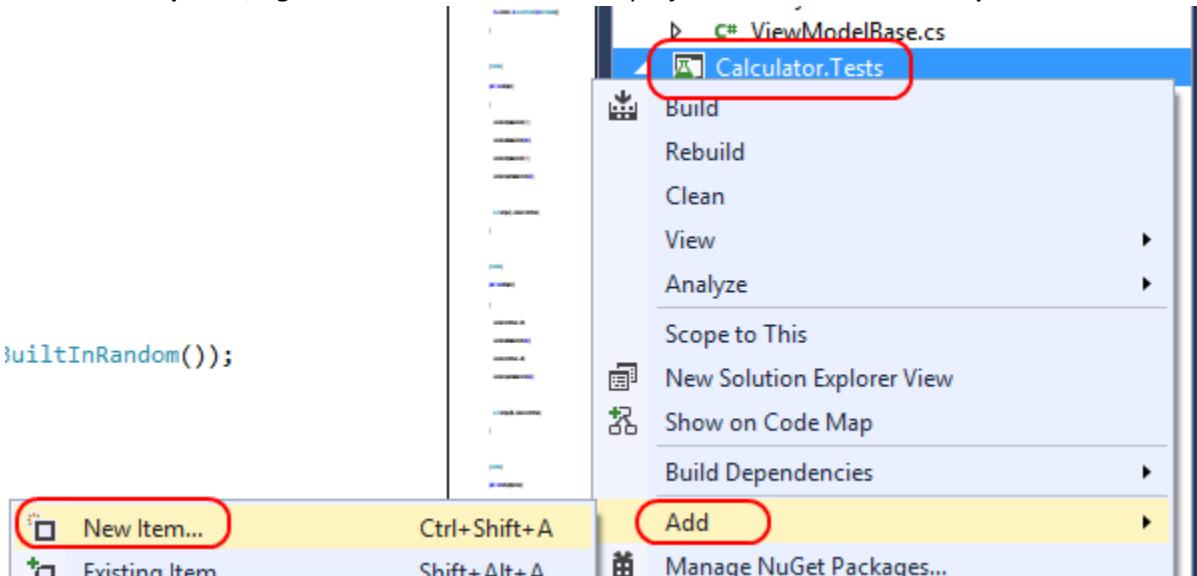
Within the body of the test, the first three lines show accessing the data from the row based on column names, which are “first”, “second”, and the expected “sum”. This will allow you to provide as many test cases as you like without having to rewrite the code based purely on changing parameters.

4. Expose a **TestContext** property by adding the following code to **MathTests.cs**. This will get set by the test engine with each row as it's ready for testing.

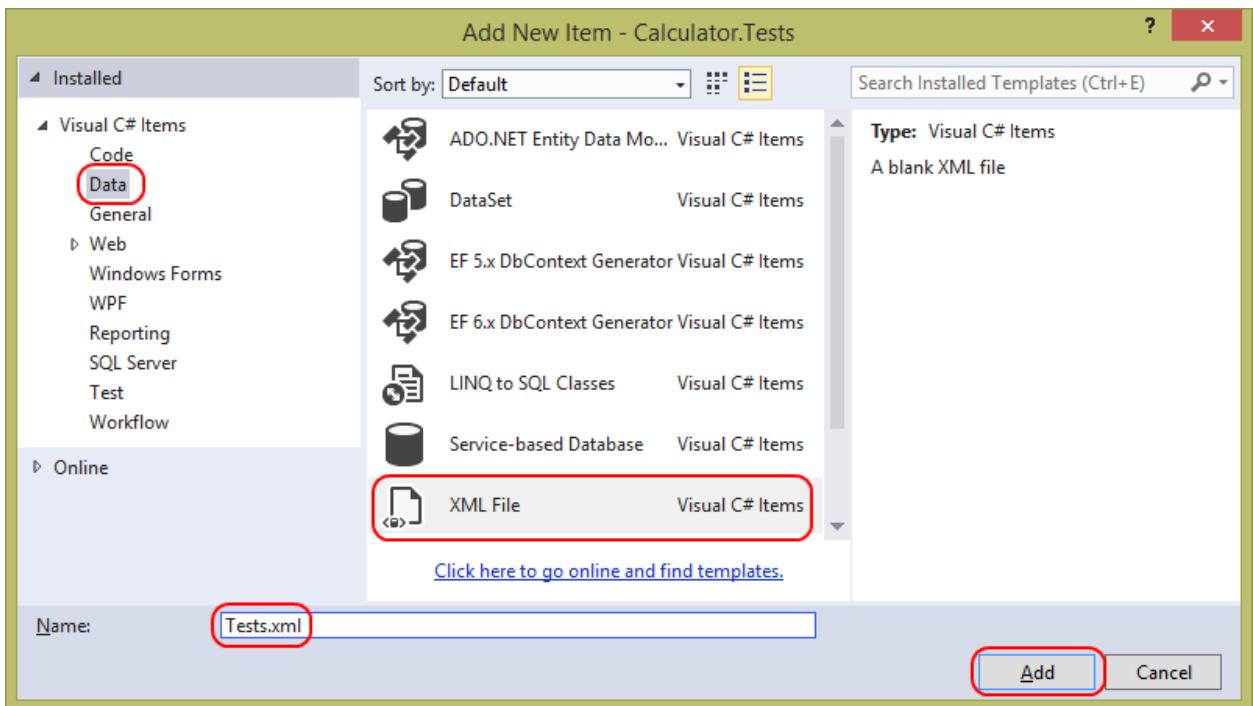
C#

```
public TestContext TestContext { get; set; }
```

5. In **Solution Explorer**, right-click the **Calculator.Tests** project node and select **Add | New Item....**



6. Select the **Visual C# Items | Data** category and the **XML File** template. Type “**Tests.xml**” as the file name and click **Add** to create.

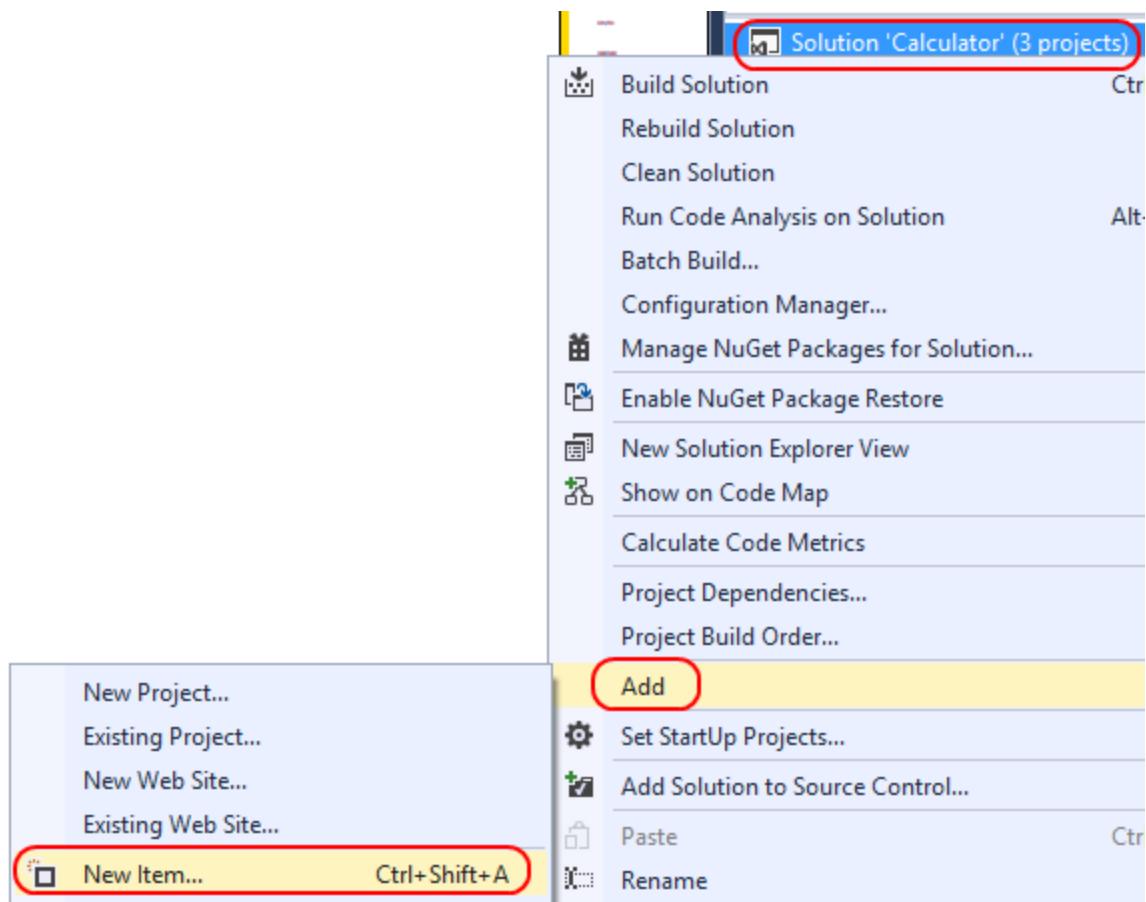


7. Add the following XML as the body of the document. When the test runs, it will iterate once with the values **1** and **2** and expect a sum of **3**. Press **Ctrl+S** to save the file.

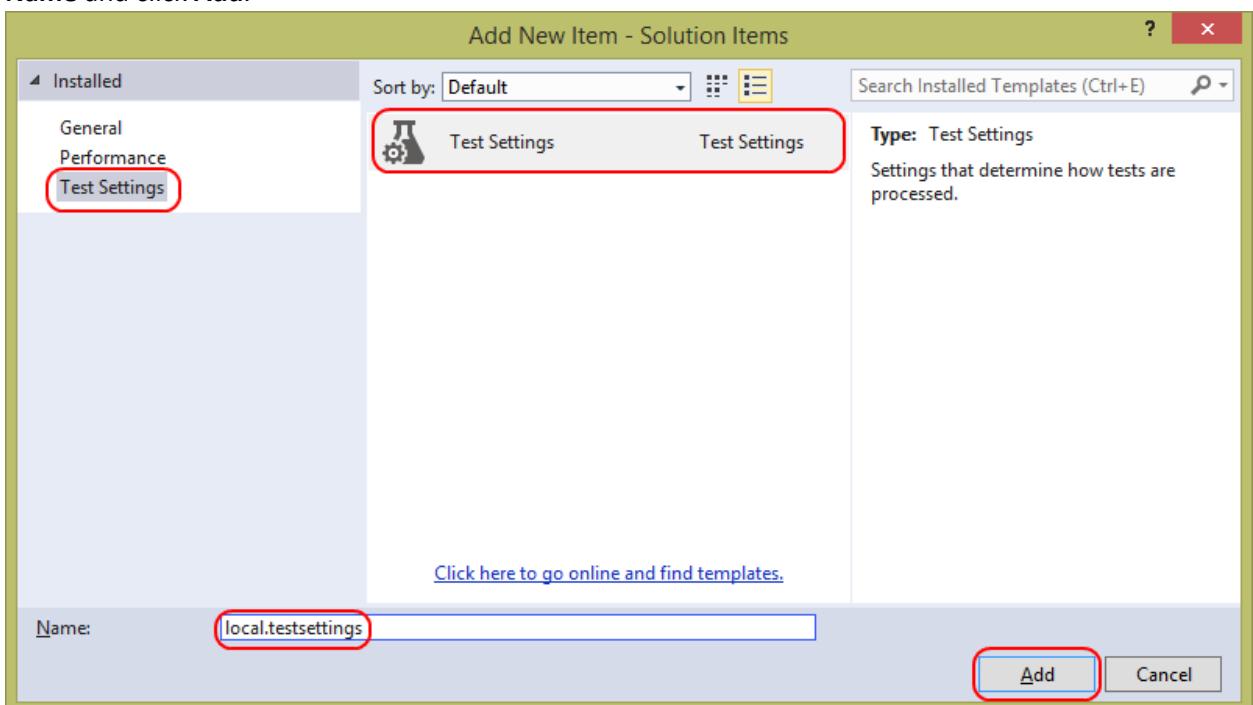
```
XML
<tests>
  <add>
    <first>1</first>
    <second>2</second>
    <sum>3</sum>
  </add>
</tests>
```

However, there is one additional step required for the test to run, which is to configure the deployment of **Tests.xml** so that the test engine can load it. For this, you will create a **Test Settings** file.

8. In **Solution Explorer**, right-click the solution node and select **Add | New Item....**

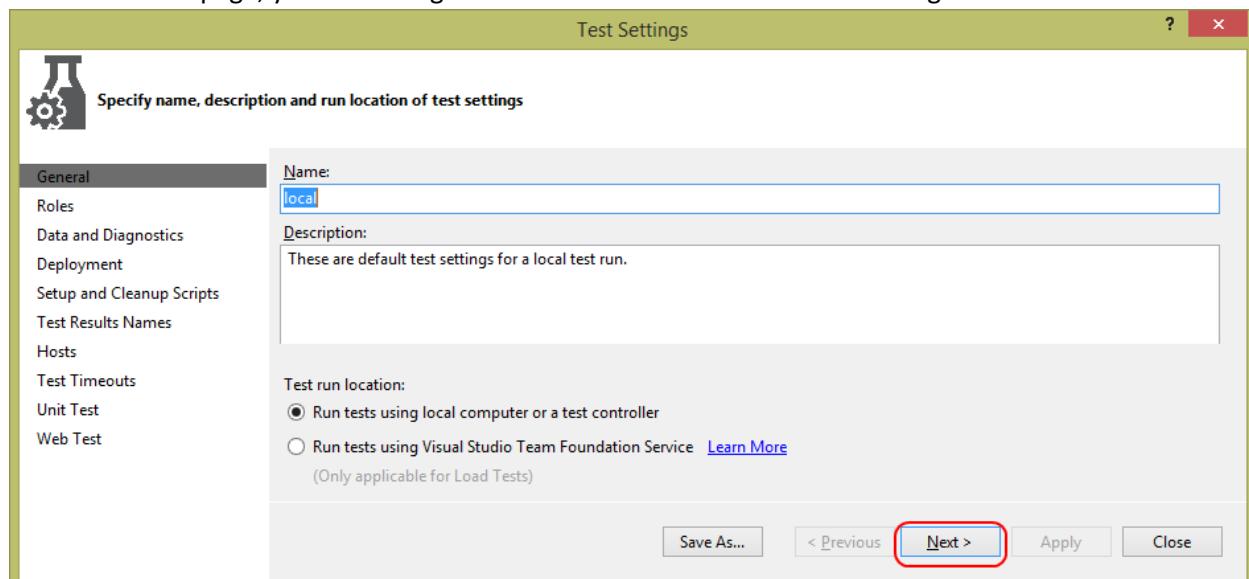


9. Select the **Test Settings** category and the **Test Settings** template. Type “**local.testsettings**” as the **Name** and click **Add**.

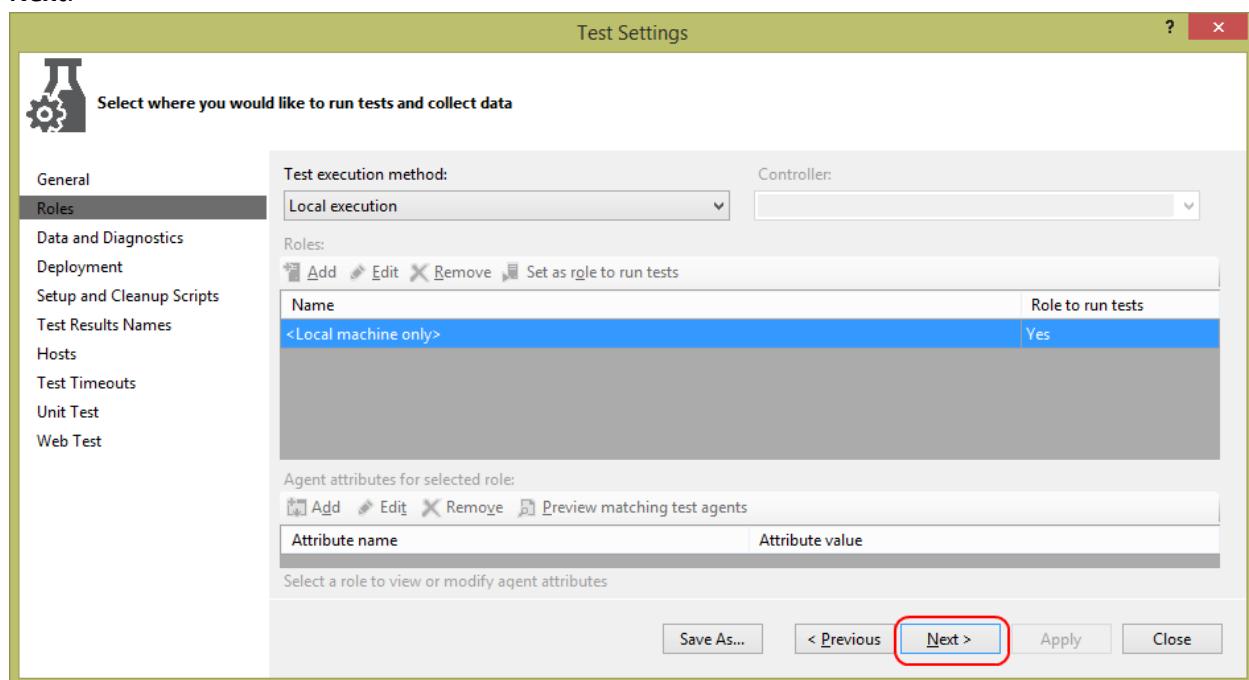


The **Test Settings** wizard contains ten steps, although there is only one step needed to configure the deployment. However, you can step through it for the purposes of this lab.

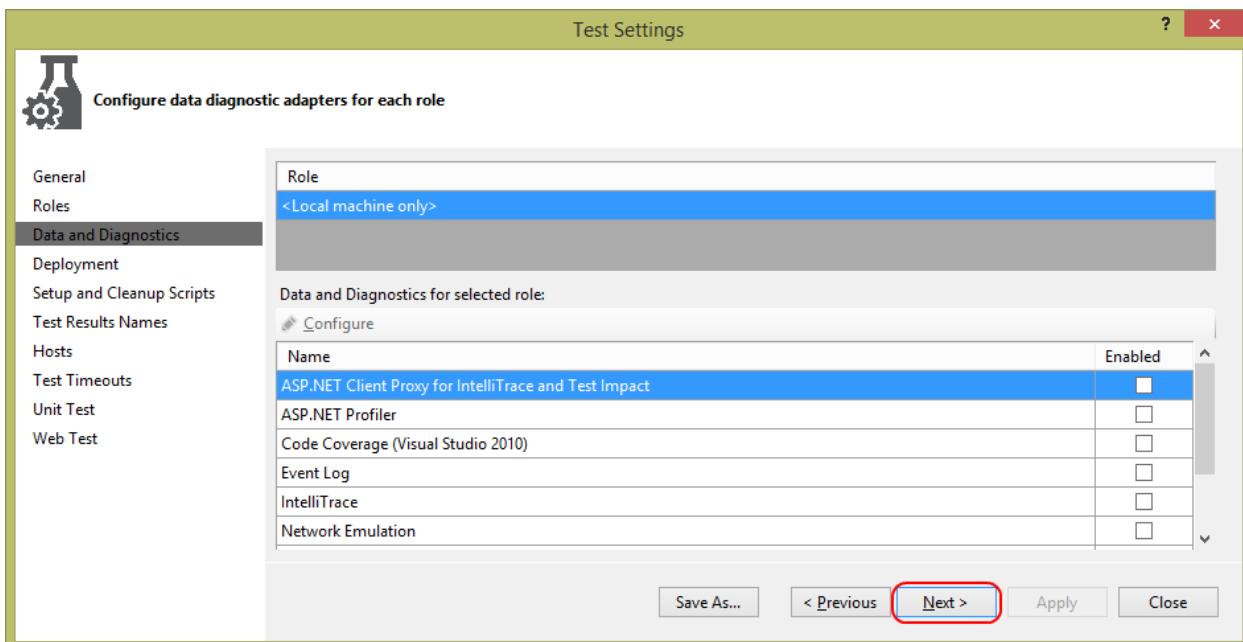
10. On the **General** page, you can configure some metadata about the test settings. Click **Next**.



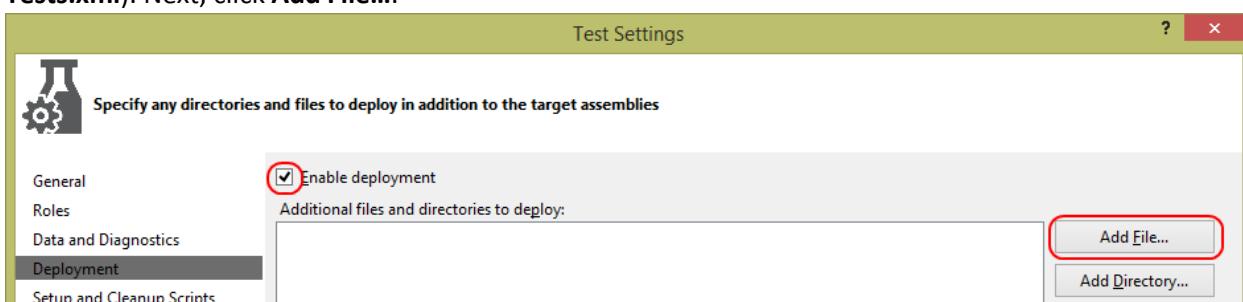
11. On the **Roles** page, you can define different roles and their involvement in running the tests. This isn't applicable here since you only plan to run unit tests that don't require anything special. Click **Next**.



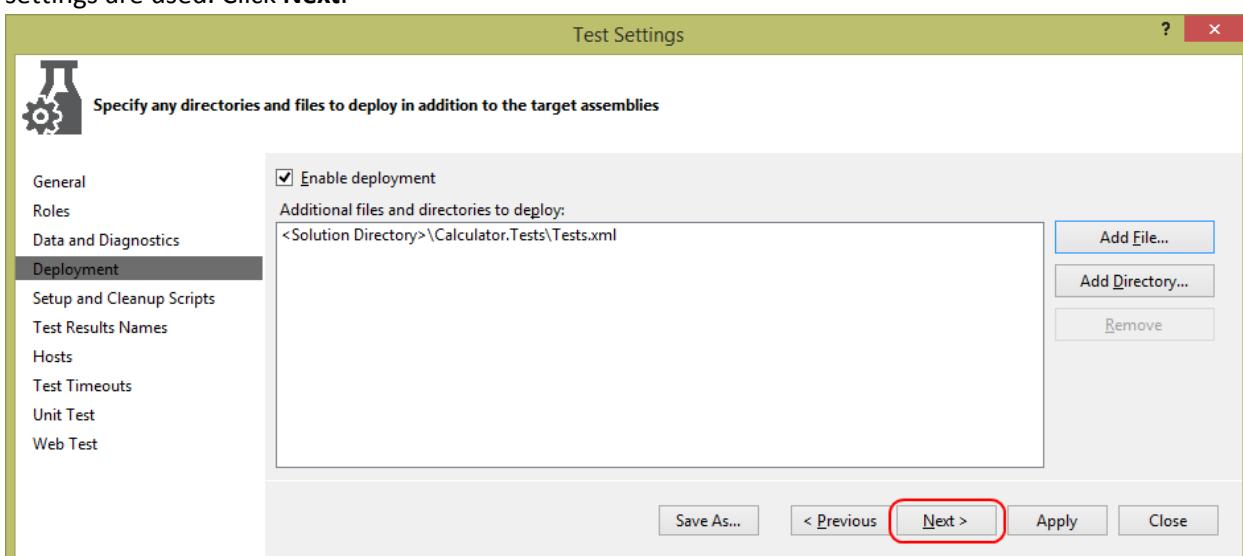
12. On the **Data and Diagnostics** page, you can configure the various ways of collecting data and diagnostics during the test run. Click **Next**.



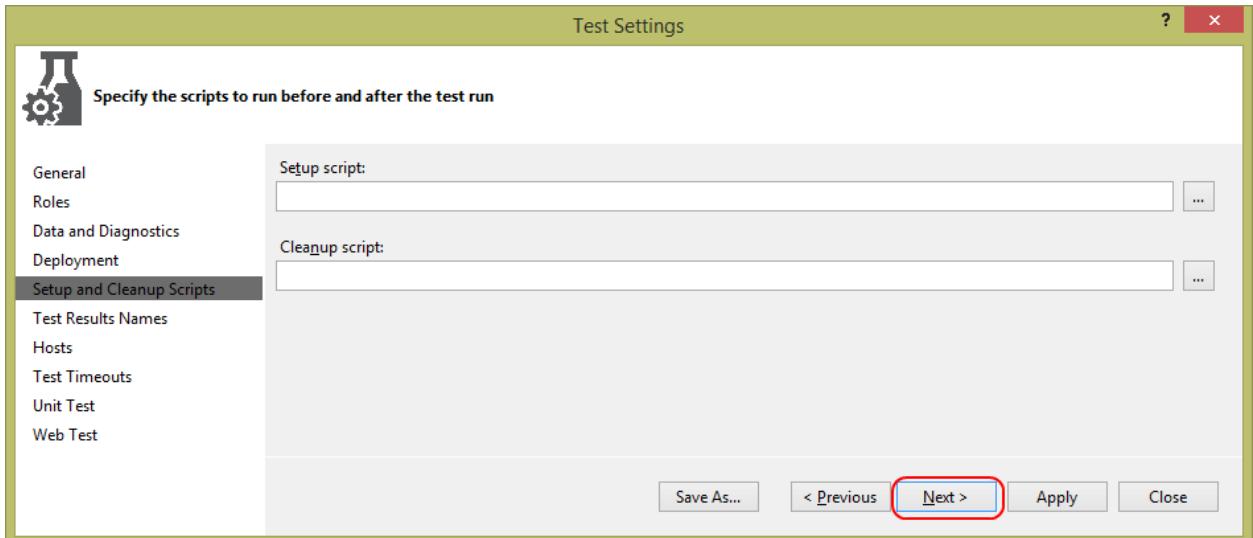
13. On the **Deployment** page, check the **Enable Deployment** box to enable deployment of files (like **Tests.xml**). Next, click **Add File....**



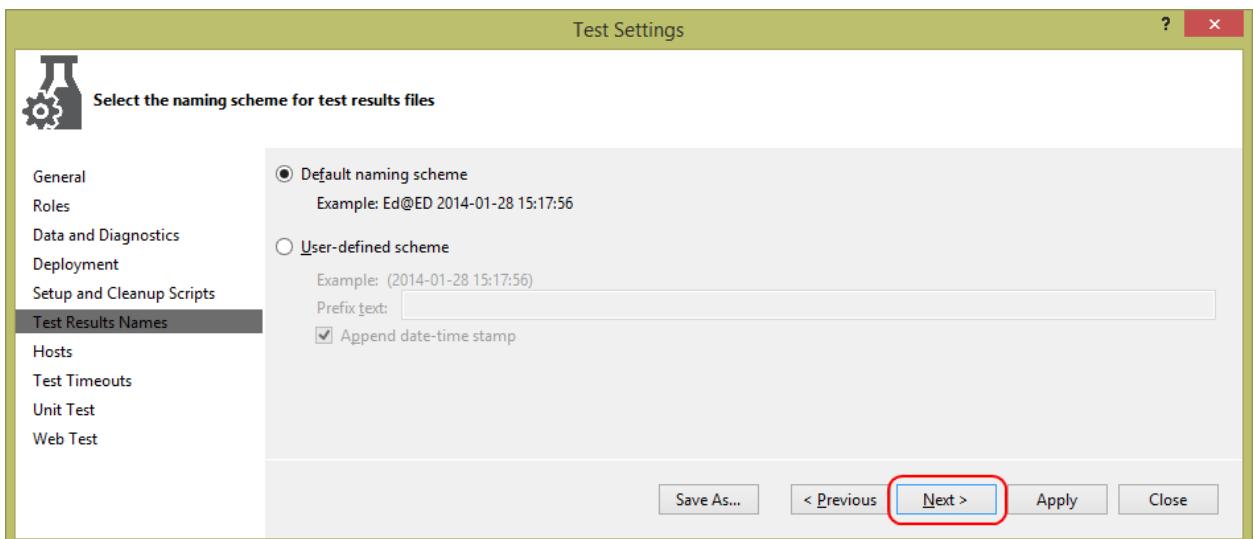
14. Locate **Tests.xml** and add it. Now it will be deployed to the test working directory when these settings are used. Click **Next**.



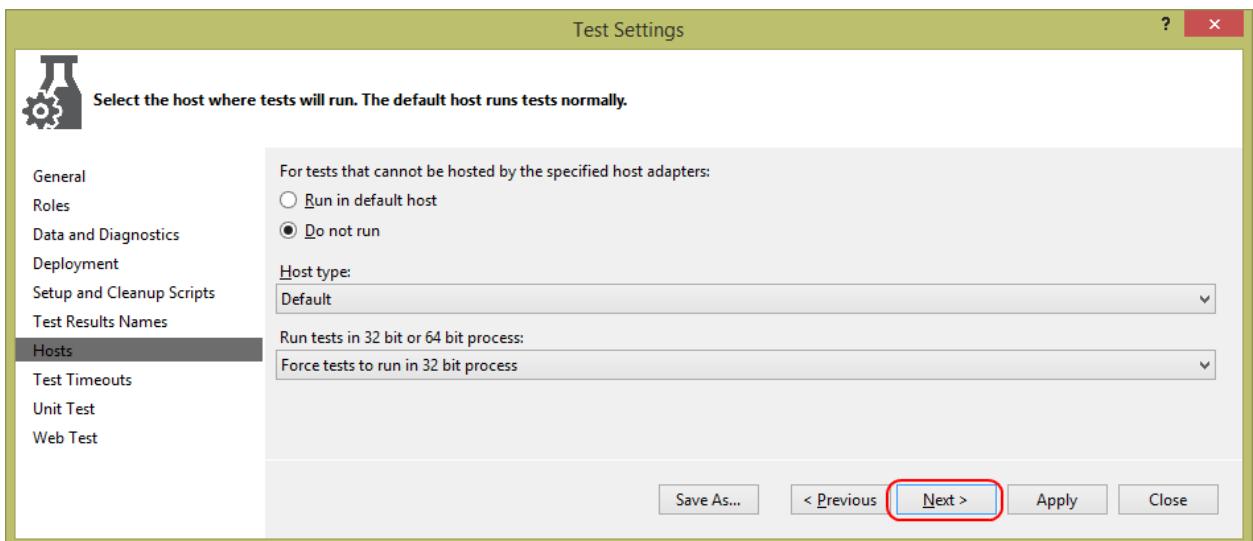
15. On the **Setup and Cleanup Script** page, you can configure scripts to be run before and after your test run. Click **Next**.



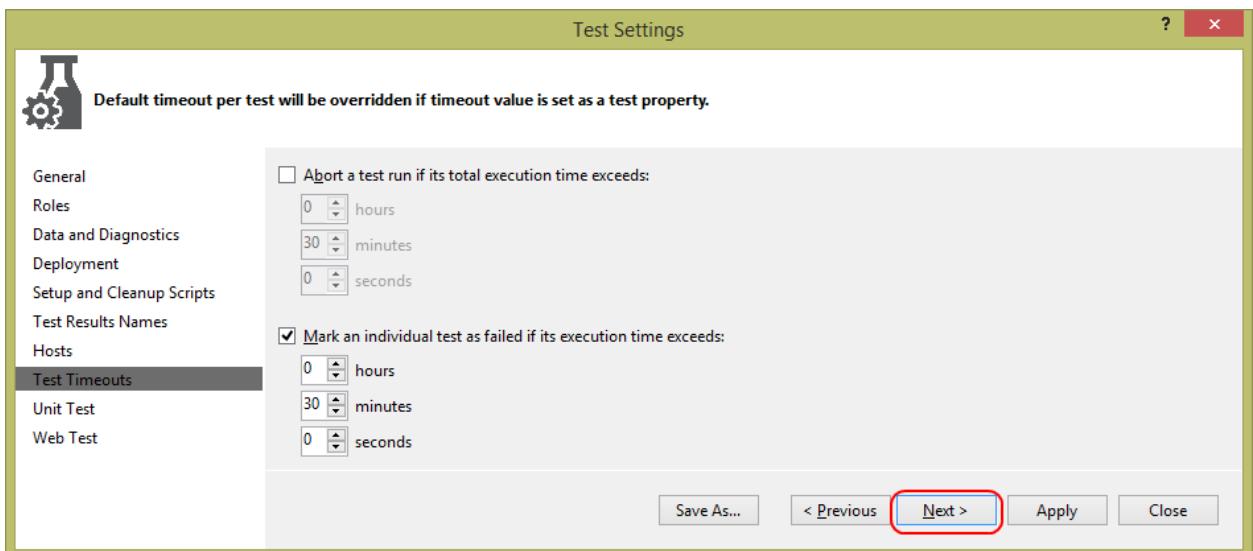
16. On the **Test Results Names** page, you can configure the naming scheme for test runs. The default name includes the user running the test, the machine the test is run on, and the date and time. Click **Next**.



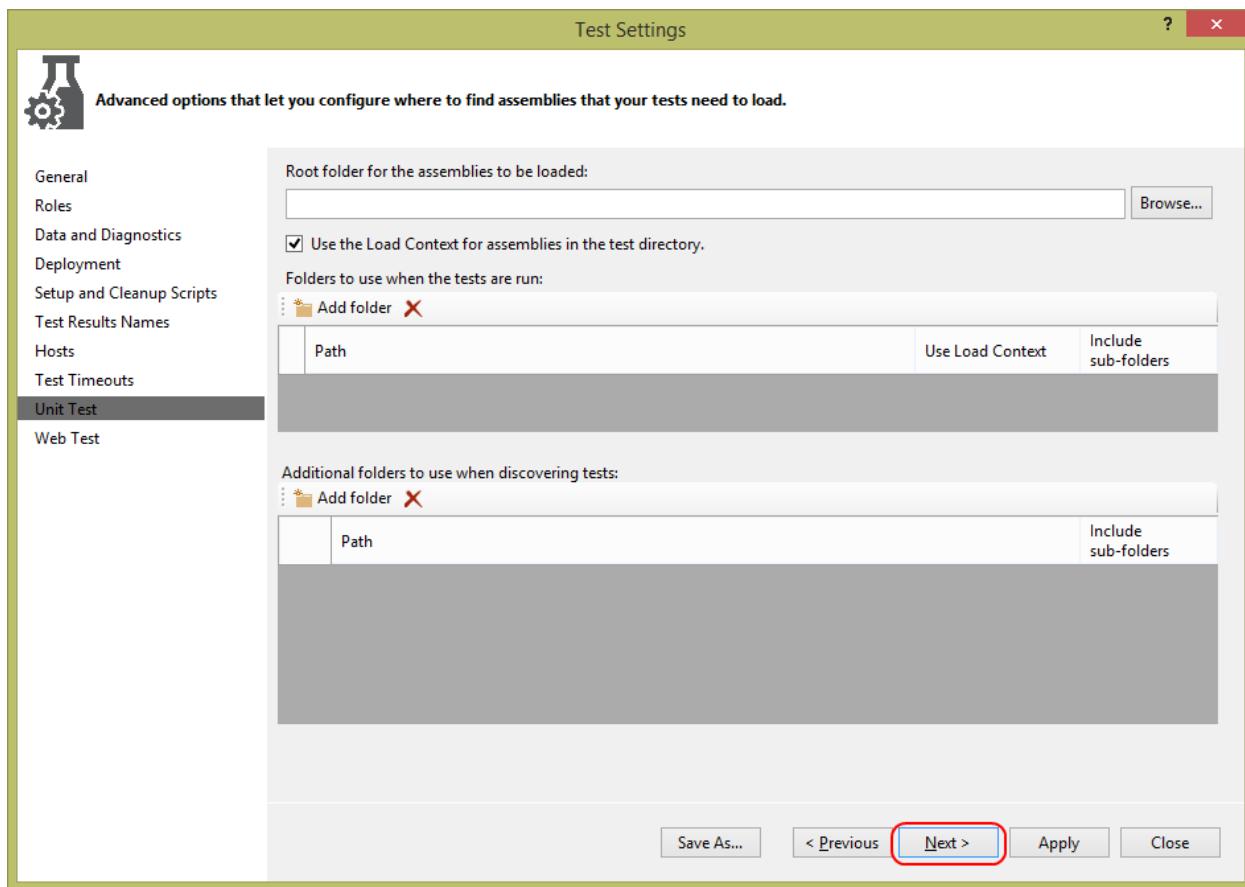
17. On the **Hosts** page, you can configure where and how the tests are run. Click **Next**.



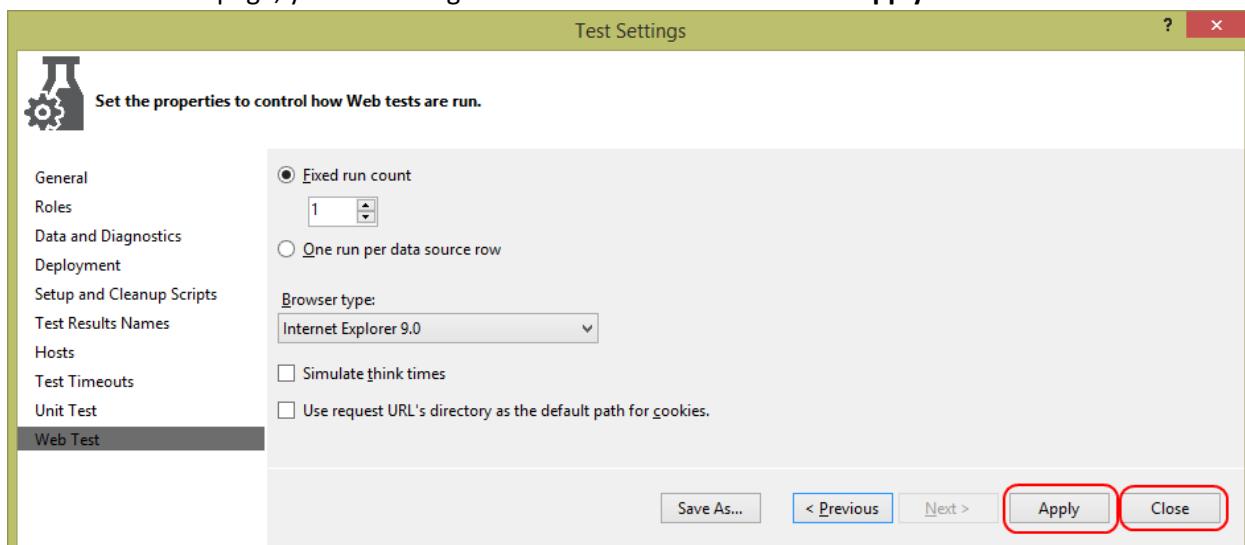
18. On the **Test Timeouts** page, you can set timeouts for each test as well as the whole test run. Click **Next**.



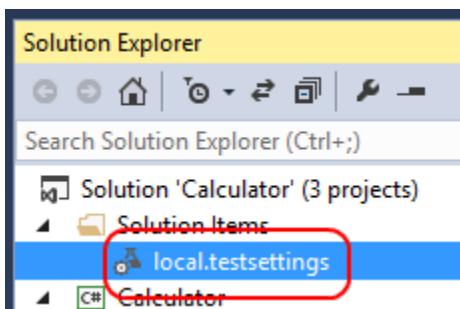
19. On the **Unit Test** page, you can configure additional folders containing reference assemblies. This allows you to have additional assemblies referenced by your tests to be kept in once place, rather than getting copied to the test working folder for each run. Click **Next**.



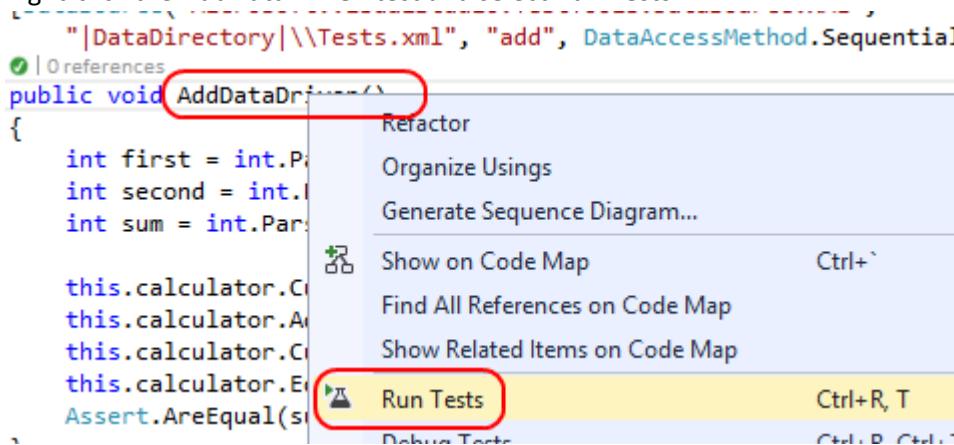
20. On the **Web Test** page, you can configure how Web tests are run. Click **Apply** and then **Close**.



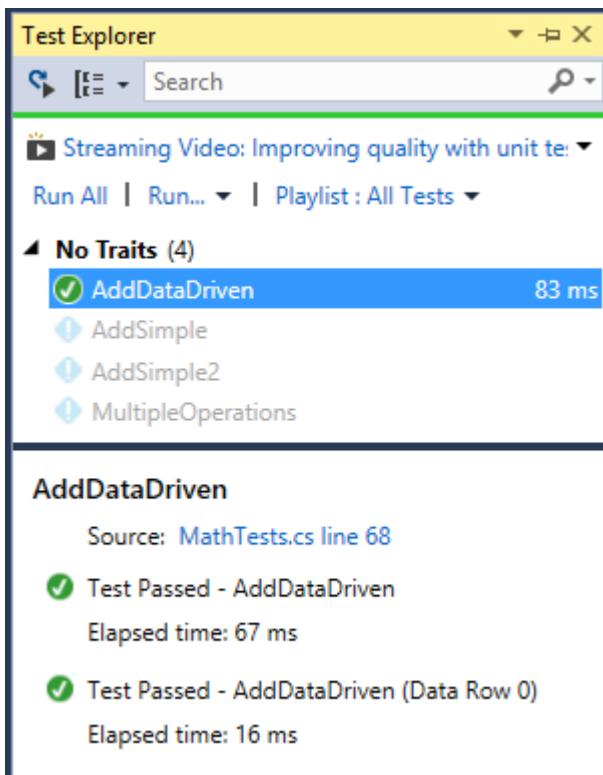
Note that the test settings file is part of the solution, and not specific to any testing project.



21. From the main menu, select **Test | Test Settings | Select Test Settings File**.
22. Select the **local.testsettings** file you just created, which will be in the root of the solution directory.
23. Right-click the **AddDataDriven** test and select **Run Tests**.



24. In **Test Explorer**, you can follow the progress of the test. Once it completes, select the test and notice how the bottom pane shows the overall success, followed by the results from each row. In this case, there is only one.

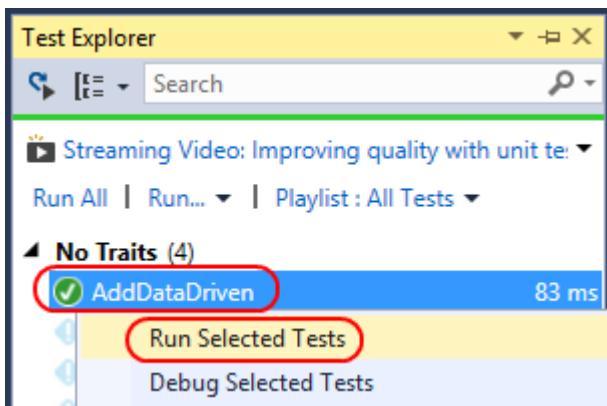


25. In **Tests.xml**, add these two additional records inside the closing tag.

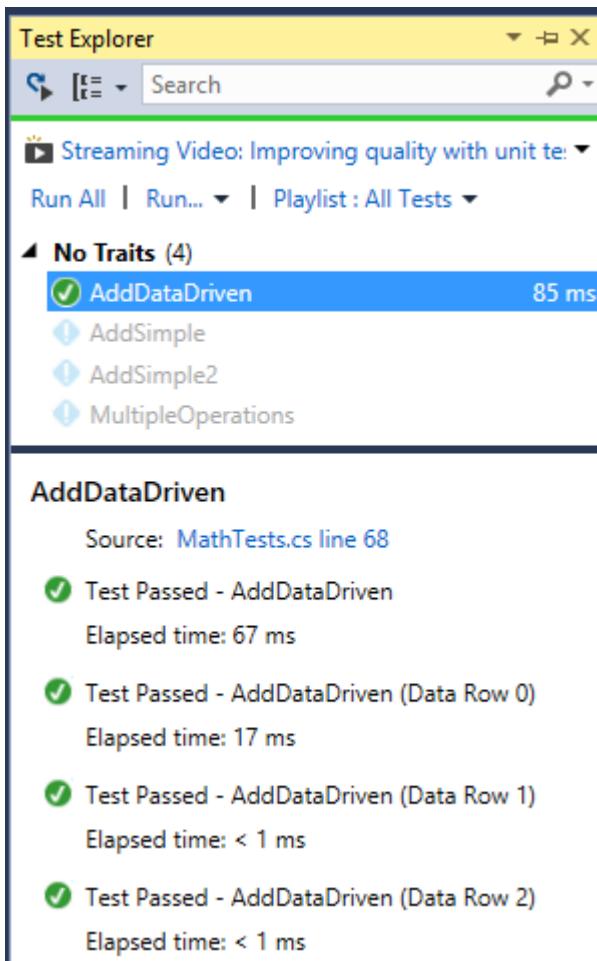
XML

```
<add>
  <first>29</first>
  <second>15</second>
  <sum>44</sum>
</add>
<add>
  <first>10000</first>
  <second>33333</second>
  <sum>43333</sum>
</add>
```

26. In **Test Explorer**, right-click **AddDataDriven** and select **Run Selected Tests**.



After the tests complete, you will notice that there are now three rows of data, and all were successful.



Task 2: Analyzing code coverage

In this task, you'll perform a code coverage analysis in order to determine which aspects of the codebase are not covered by unit tests. By identifying the areas of code that are not being exercised during unit testing, you can prioritize the addition of new tests for those areas. However, be

cautious of relying too much on the metrics provided by code coverage alone. While it is a great tool for understanding the reach of your tests, it does not necessarily indicate that all scenarios are being properly tested.

1. From the main menu, select **Test | Analyze Code Coverage | All Tests**. This will build and run all tests.
2. Locate the **Code Coverage Results** panel, which is probably docked to the bottom of Visual Studio. If you expand **calculator.dll | Calculator | CalculatorViewModel**, you can see that there are quite a few methods that are completely uncovered by these tests. Notice how each parent row aggregates the statistics of its children. This gives you a great way to quickly determine the overall code coverage in a class, namespace, or assembly. Double-click the **Back(object)** method to navigate to it.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Ed_ED 2014-01-29 09_31_03.cove...	59	21.45 %	216	78.55 %
calculator.dll	59	28.50 %	148	71.50 %
Calculator	59	28.50 %	148	71.50 %
BuiltInRandom	3	60.00 %	2	40.00 %
CalculatorViewModel	45	26.95 %	122	73.05 %
<.ctor>b_2(objec...)	4	100.00 %	0	0.00 %
Add(object)	0	0.00 %	5	100.00 %
AddHours(object)	5	100.00 %	0	0.00 %
AddMinutes(object)	5	100.00 %	0	0.00 %
AddRandom(obje...)	4	100.00 %	0	0.00 %
AddSeconds(obje...)	5	100.00 %	0	0.00 %
AppendKeys(obje...)	2	14.29 %	12	85.71 %
Back(object)	7	100.00 %	0	0.00 %
Calculate()	0	0.00 %	20	100.00 %

3. In the **Back** method, you can see how the code is shaded red. These lines were not exercised as part of the last test run.

```
private void Back(object _)
{
    if (this.CurrentValue == 0)
    {
        return;
    }

    this.CurrentValue = this.CurrentValue / 10;
}
```

4. In **MathTests.cs**, add the following test. It will exercise the **Back** method via its command.

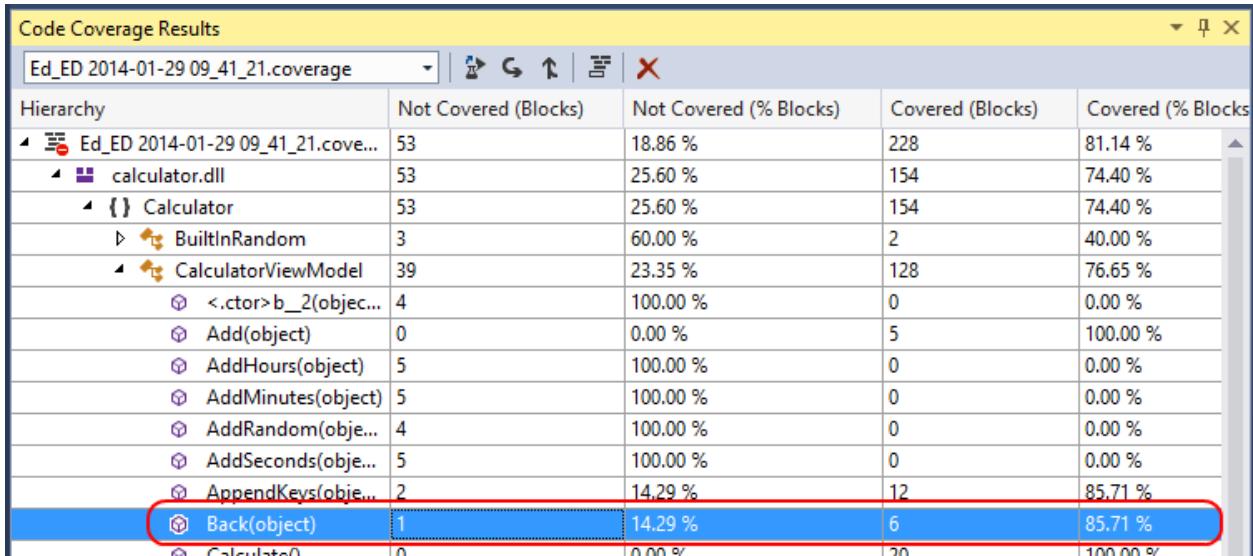
```
C#
[TestMethod]
public void Back()
{
    calculator.CurrentValue = 100;
    calculator.BackCommand.Execute(null);
```

```

        Assert.AreEqual(10, calculator.CurrentValue);
    }
}

```

5. From the main menu, select **Test | Analyze Code Coverage | All Tests**. Note that analyzing code coverage invokes a test run, so you don't need to do that separately.
6. In the **Code Coverage Results**, expand the new run out until you find the **Back** method again. Now you can see that it went from having **6** untested lines to only **1**.



7. Switch back to **CalculatorViewModel.cs** to see the updated code coverage highlighting. It's now easy to see which lines have been tests (blue) against the one that was not (red).

```

1 reference
private void Back(object _)
{
    if (this.CurrentValue == 0)
    {
        return;
    }

    this.CurrentValue = this.CurrentValue / 10;
}

```

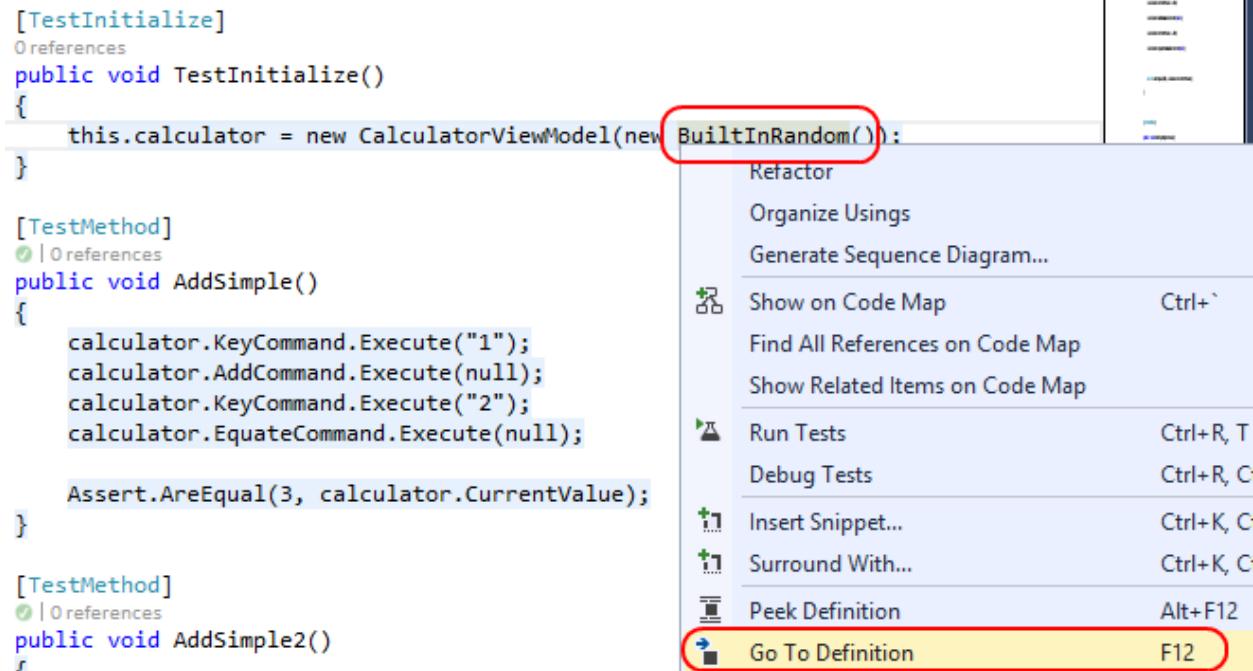
Task 3: Using Microsoft Fakes, stubs, & shims

In this task, you'll add a Microsoft Fakes assembly. Microsoft Fakes help you isolate the code you are testing by replacing other parts of the application with stubs or shims. These are small pieces of code that are under the control of your tests. By isolating your code for testing, you know that if the test fails, the cause is there and not somewhere else. Stubs and shims also let you test your code even if other parts of your application are not working yet. Note that Microsoft Fakes requires Visual Studio Ultimate or Visual Studio Premium.

1. In the **Code Coverage Results** panel, note that the **AddRandom** method has no code coverage.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
AddRandom(object)	4	100.00 %	0	0.00 %

2. Locate the **TestInitialize** method in **MathTests.cs**. This is where the **CalculatorViewModel** is created for use in these tests. Fortunately, the developer who built this class used a dependency injection model, so you can specify which class is providing the random number generation. Right-click **BuiltInRandom** and select **Go To Definition**.



3. **BuiltInRandom** is a fairly simple class. The one method it exposes generates a random number from 1-100. However, there's a potential problem here since the number is unpredictable for test purposes, so there's no good way to rely on this class for **Assert** calls since you won't know what number it's returning.

```

2 references
public class BuiltInRandom : IRandom
{
    private Random _random = new Random();

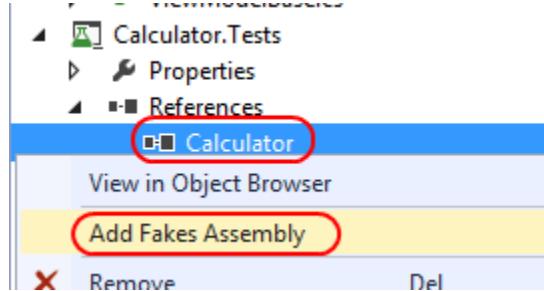
    2 references
    public int GetRandomNumber()
    {
        return this._random.Next(100) + 1;
    }
}

```

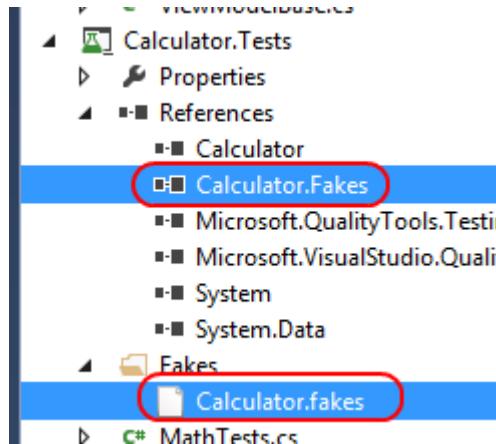
Fortunately, the developer had the foresight to use a dependency injection model when defining **CalculatorViewModel**, so you can simply create your own **IRandom** that returns a hardcoded value and pass that in as a parameter. However, this could get messy if there turns out to be a need for different values being returned. Fortunately, this is where **stubs** come in. By adding a Microsoft

Fakes assembly for the **Calculator** library, you can quickly and easily “stub out” a valid **IRandom** that you control with a method defined in line with the rest of the code.

4. In **Solution Explorer**, right-click the **Calculator** assembly node under the **References** of **Calculator.Tests** and select **Add Fakes Assembly**.



5. This will generate a **Calculator.Fakes** assembly that has helper classes that allow you to override (or replace) methods and properties of the original types. It also makes it very easy to implement versions of interface classes without having to define all the required methods and properties.



6. At the top of **MathTests.cs**, add the following **using** declaration.

```
C#
using Calculator.Fakes;
```

7. Add the following test to **MathTests.cs**.

```
C#
[TestMethod]
public void RandomSimple()
{
    StubIRandom random = new StubIRandom();
    random.GetRandomNumber = () => { return 10; };

    CalculatorViewModel myCalculator = new CalculatorViewModel(random);
    myCalculator.CurrentValue = 100;
    myCalculator.AddRandomCommand.Execute(null);
```

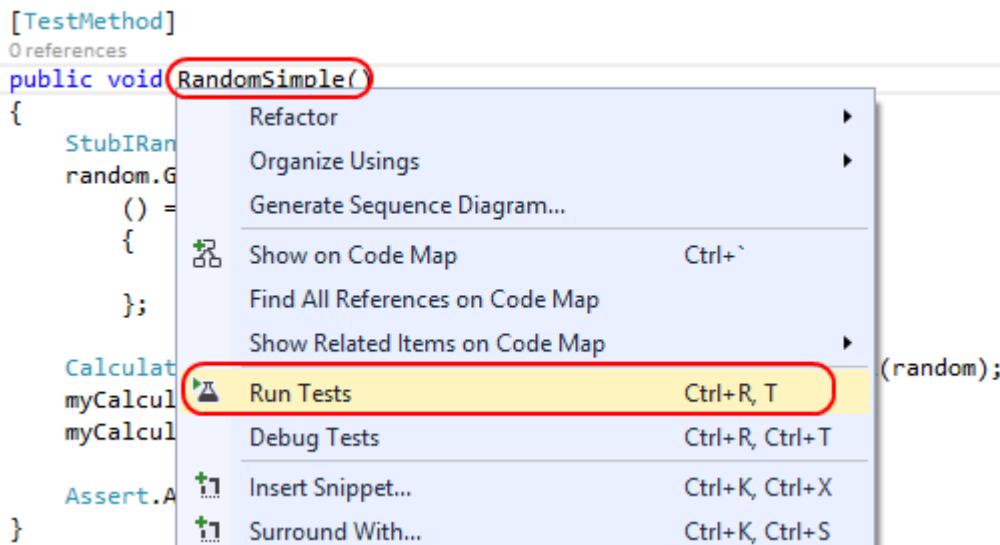
```

        Assert.AreEqual(110, myCalculator.CurrentValue);
    }
}

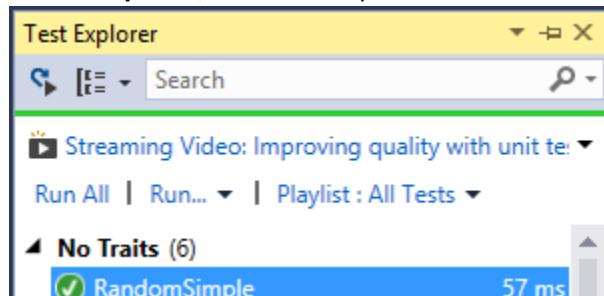
```

The magic in this code lives in the first two lines. First, you're creating a **StubIRandom**, which was generated by Fakes. Next, you're setting its **GetRandomNumber** method to an anonymous delegate that always returns **10**. The rest of the test is fairly straightforward code that creates a new **CalculatorViewModel**, sets its initial value to **100**, and then adds a "random" number to it. However, you've hardcoded that random number to **10**, so you know it'll result in **110** every time.

- Right-click **RandomSimple** and select **Run Tests**.



In **Test Explorer**, the test will proceed and succeed.



While stubs are great for extending classes and implementing interfaces, sometimes there is a need to override behavior in places you don't have easy access to, such as system calls and static methods. For this, Microsoft Fakes provides **shims**.

- In the **Code Coverage Results** panel, locate the **AddSeconds** method. Since it doesn't have any code coverage at this point, double-click it to view the code so you can determine the best way to write a test.

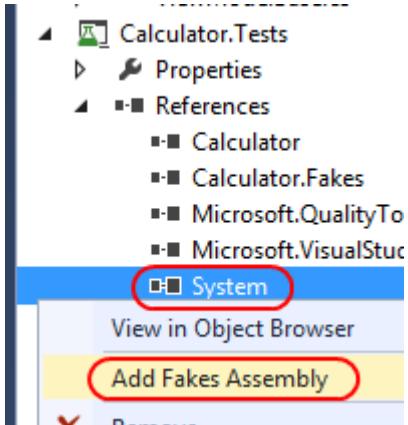
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
⌚ AddHours(object)	5	100.00 %	0	0.00 %
⌚ AddMinutes(object)	5	100.00 %	0	0.00 %
⌚ AddRandom(obje...)	4	100.00 %	0	0.00 %
⌚ AddSeconds(obje...)	5	100.00 %	0	0.00 %
⌚ AppendKeystoobj...	2	14.29 %	12	85.71 %

While the code for **AddSeconds** is simple, it unfortunately introduces a problem. Unlike the random functionality from earlier, the developer did not abstract out the time sourcing or provide it as a dependency that could be injected. In a perfect world, you might go into the code and refactor it to use better practices, but for the purposes of this lab, you'll need to test it as-is.

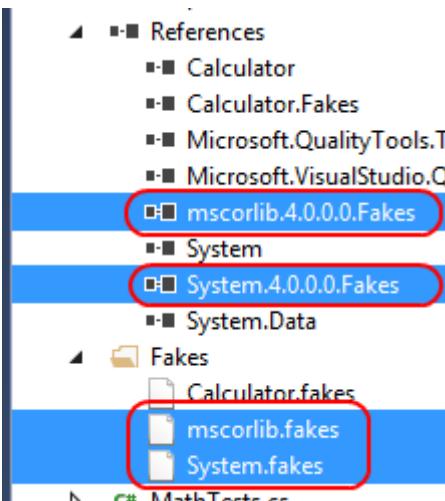
```
1 reference
private void AddSeconds(object _)
{
    this.CurrentValue += DateTime.Now.Second;
}
```

In order to test this method, you'll ultimately need to control what **DateTime.Now** returns, which is a tall order, considering that it's a type that's built into the framework. However, you can create a Fakes assembly for the **System** assembly, which will allow you to shim this property for the scope of the method.

10. In **Solution Explorer**, right-click the **System** assembly node under the **References** of **Calculator.Tests** and select **Add Fakes Assembly**.



Similar to before, Visual Studio creates a Fakes assembly for **System**. Note that it also creates one for **mscorlib**.



11. At the top of **MathTests.cs**, add the following two **using** directives. The first references the new Fakes assembly. The second provides access to the **ShimsContext** class, which is used to scope the period during which a shim applies.

```
C#
using System.Fakes;
using Microsoft.QualityTools.Testing.Fakes;
```

12. Add the following test to **MathTests.cs**.

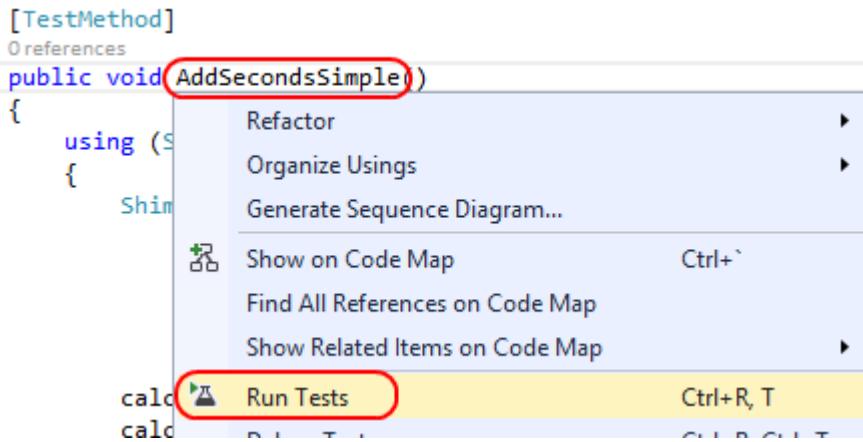
```
C#
[TestMethod]
public void AddSecondsSimple()
{
    using (ShimsContext.Create())
    {
        ShimDateTime.NowGet =
            () =>
        {
            return new DateTime(2013, 3, 15, 5, 10, 15);
        };

        calculator.CurrentValue = 100;
        calculator.AddSecondsCommand.Execute(null);

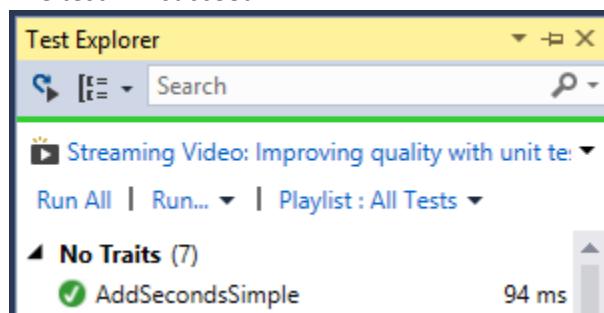
        Assert.AreEqual(115, calculator.CurrentValue);
    }
}
```

Note how the test is now enclosed within a **ShimsContext.Create** using block. This ensures that the **ShimDateTime.NowGet** method that overrides the property getter for **DateTime.Now** only applies until the object it returns is disposed at the end of the using block. The rest of the test is pretty standard, and it will work because you are controlling exactly what gets returned from **DateTime.Now**.

13. Right-click **AddSecondsSimple** and select **Run Tests**.



The test will succeed.



14. From the main menu, select **Test | Analyze Code Coverage | All Tests**. Now you can see that the test coverage has significantly improved, thanks to Microsoft Fakes.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
⌚ AddRandom(obje...	0	0.00 %	4	100.00 %
⌚ AddSeconds(obje...	0	0.00 %	5	100.00 %

Exercise 4: Continuous integration testing with Visual Studio Online

In this exercise, you'll learn about how to set up a continuous integration build & test using Visual Studio Online.

Task 1: Configuring a Visual Studio Online team project

In this task, you'll create a Visual Studio Online team project. This project will ultimately house the code for your project and will later be configured to run the unit tests automatically after a check in. This task will focus on setting up the team project and checking in the current solution.

1. When you created your **Visual Studio Online** account, you set up a domain at visualstudio.com. Navigate to that domain and log in. The URL will be something like <https://MYSITE.visualstudio.com>.
2. Under the **Recent projects & teams** panel, click the **New** link.

Recent projects & teams

New [Browse](#)

3. In the **Create new team project** dialog, type “**Unit testing**” as the **Project name** and click **Create project**. It may take a few minutes for your project to be created. Note that you can use any name you like for this project, although the screenshots will not match.

CREATE NEW TEAM PROJECT x

Project name x
Note: You cannot change the name of your project after you have created it

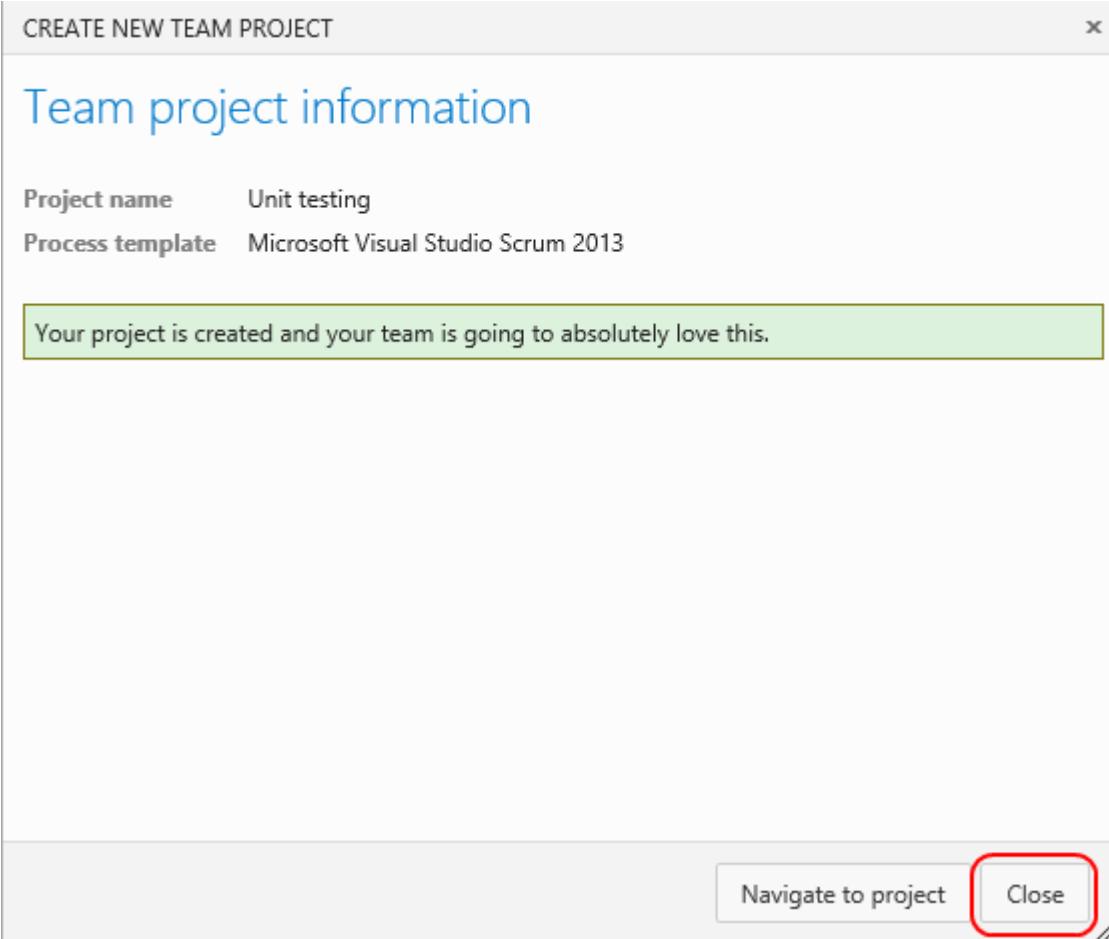
Description

Process template Microsoft Visual Studio Scrum 2013 ▼
This template is for teams who follow the Scrum methodology and use Scrum terminology.

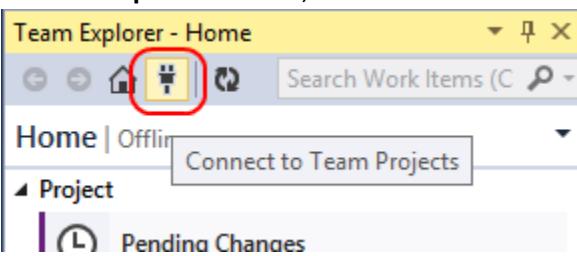
Version control Team Foundation Version Control ▼
Team Foundation Version Control (TFVC) uses a single, centralized server repository to track and version files. Local changes are always checked in to the central server where other developers can get the latest changes.

Create project Cancel

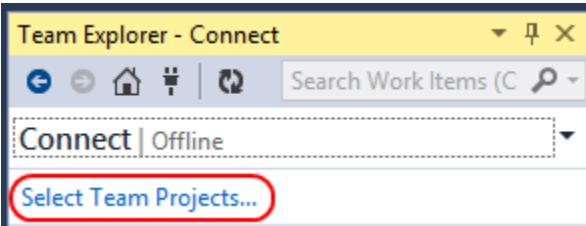
4. After the project has been created, click **Close**.



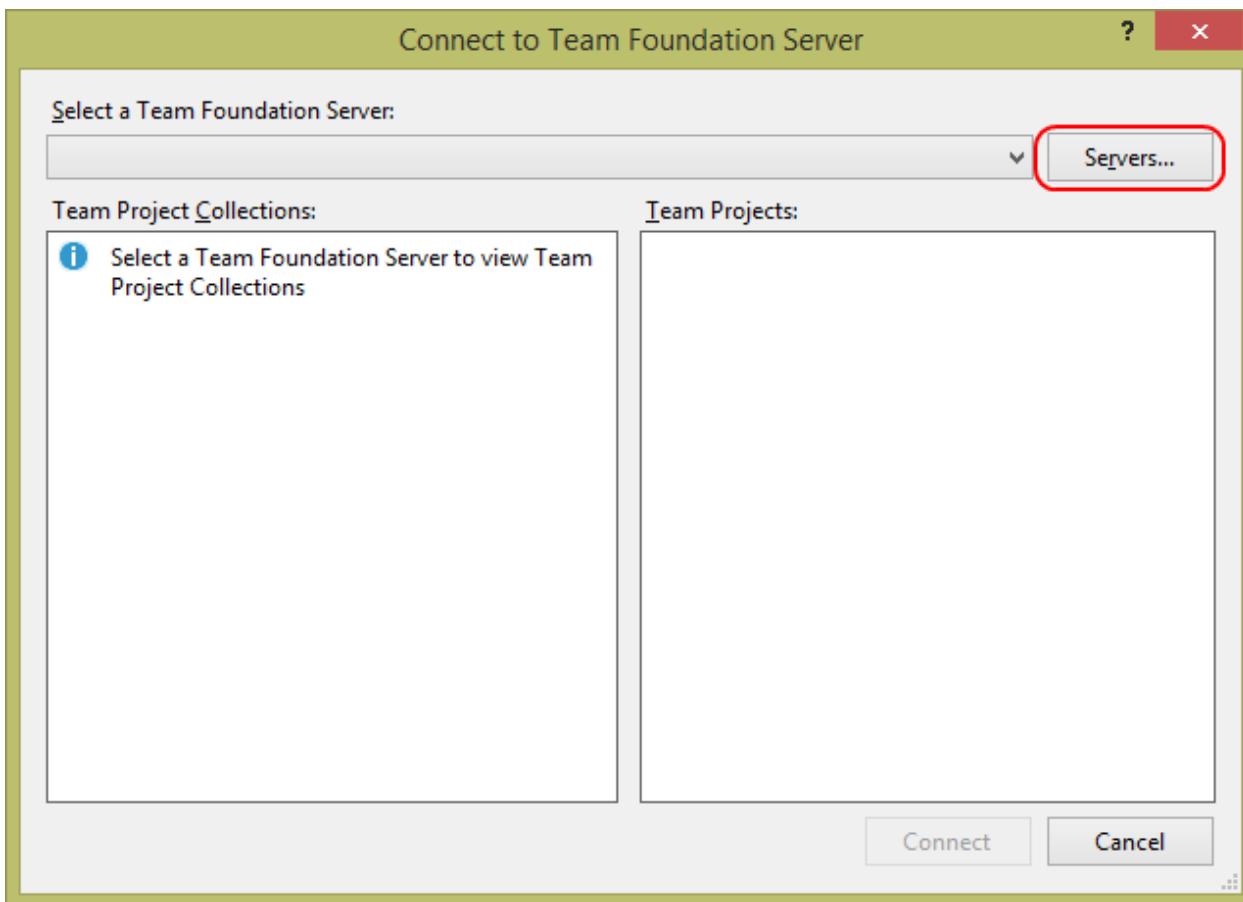
5. In **Team Explorer - Home**, click the **Connect to Team Projects** button.



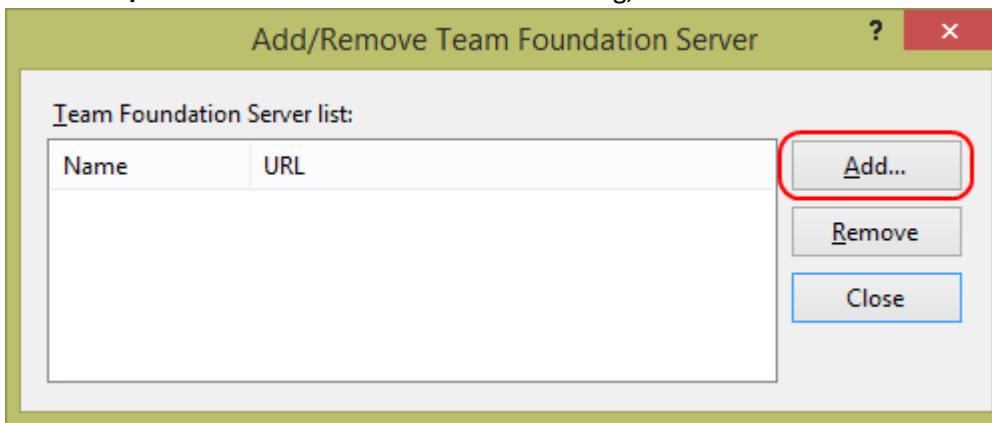
6. Next, click **Select Team Projects....**



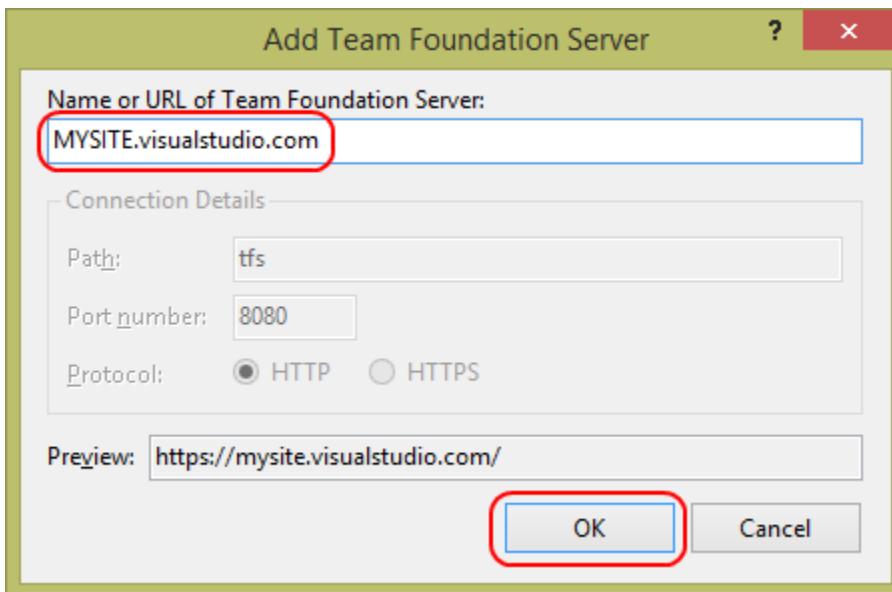
7. In the **Connect to Team Foundation Server** dialog, click **Servers....**



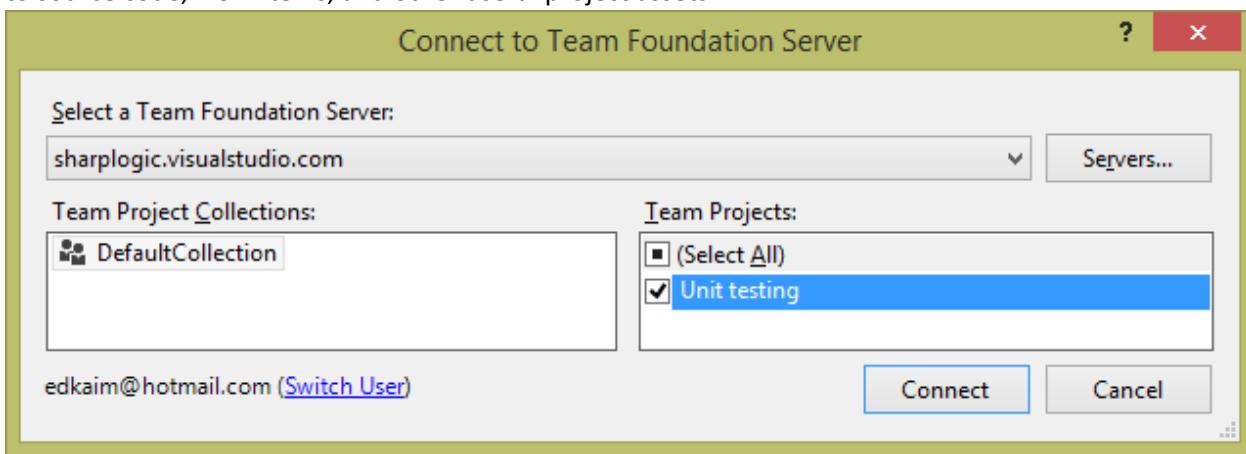
8. In the **Add/Remove Team Foundation Server** dialog, click **Add....**



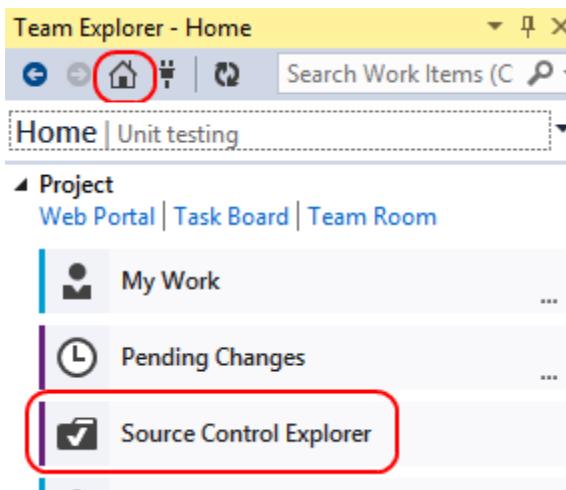
9. In the **Add Team Foundation Server** dialog, type your **visualstudio.com** domain as the **Name or URL of Team Foundation Server** and click **OK**. If asked to sign in, use the same Microsoft account you used to create the **Visual Studio Online** account.



10. In the **Connect to Team Foundation Server** dialog, check the **Unit testing** project and click **Connect**. Visual Studio will now connect to your project and perform some basic configuration to allow access to source code, work items, and other useful project assets.

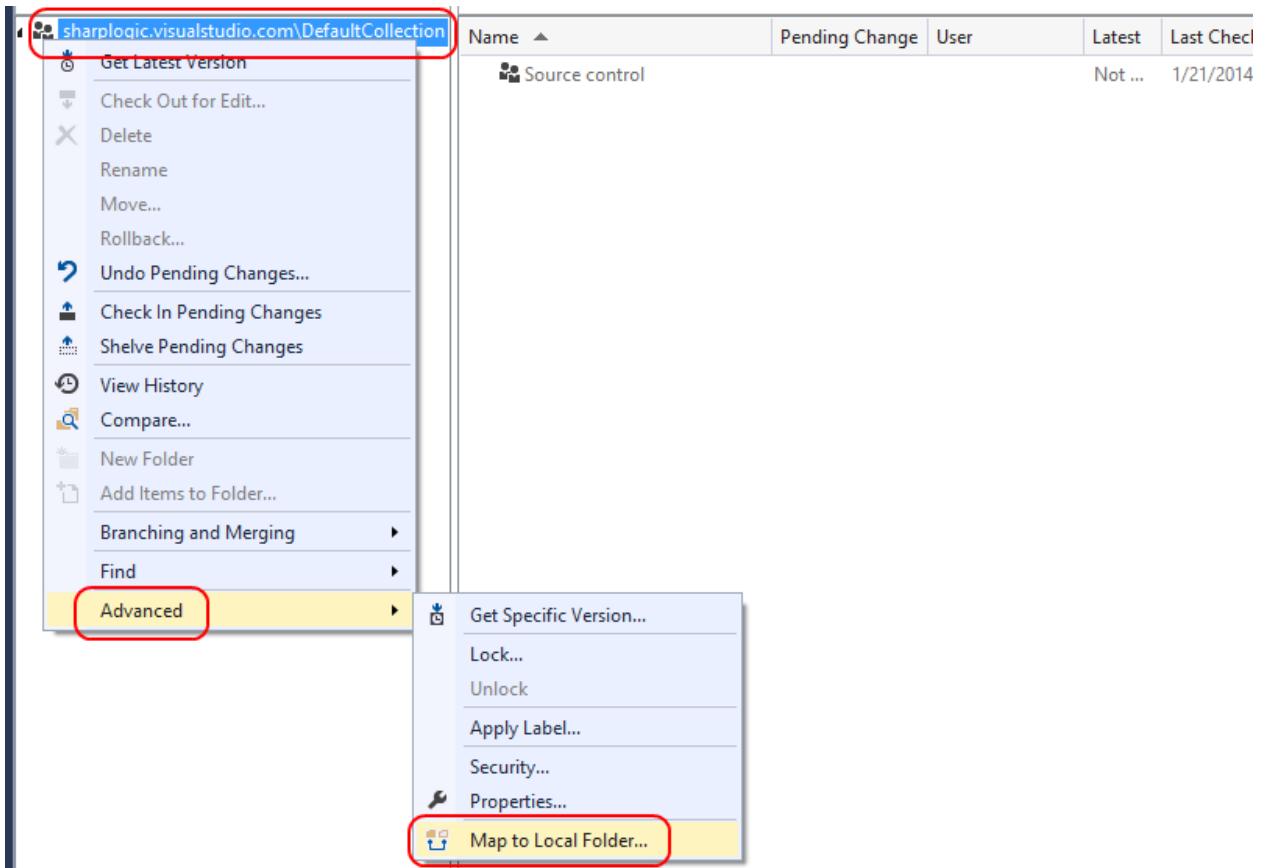


11. In **Team Explorer**, there are now several options for interacting with the newly created project. Click the **Home** button followed by **Source Control Explorer** to see the source code.

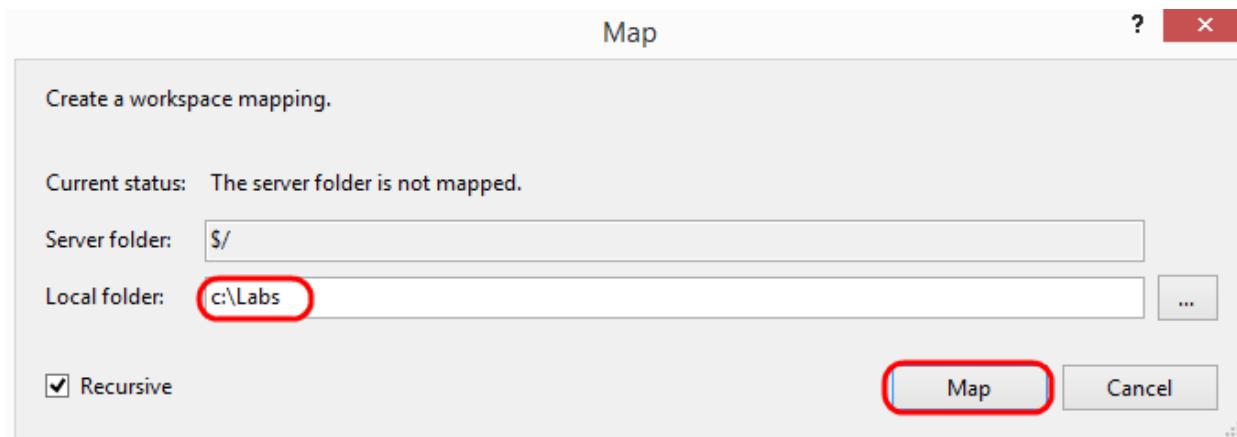


For a new project, the source code repository is empty, with the exception of some default build process templates.

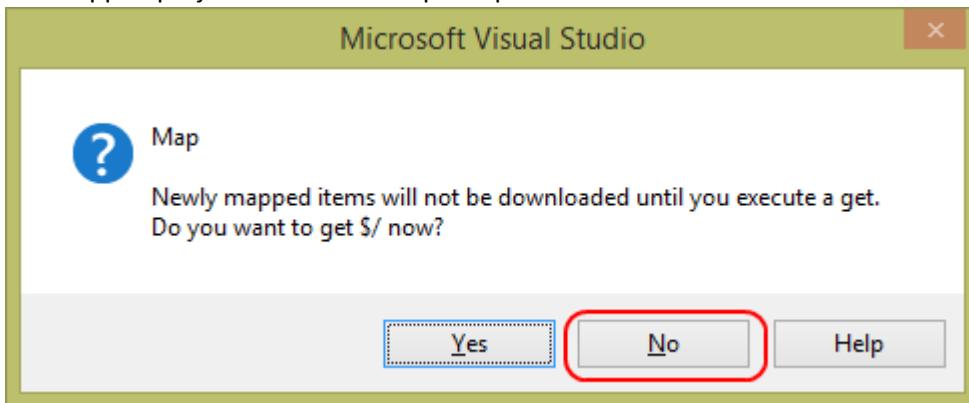
12. In **Source Control Explorer**, right-click the root collection node and select **Advanced | Map to Local Folder....**



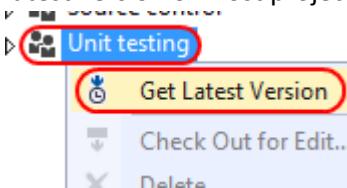
13. In the **Map** dialog, select a local folder where you'd like to keep all your source. Click **Map** to complete the mapping.



14. After mapping, Visual Studio will provide the option to download the latest version of all files within the mapped project. Click **No** to skip this process for now.



15. Right-click the **Unit testing** project and select **Get Latest Version**. This will be the way you get the latest version of most projects and files.

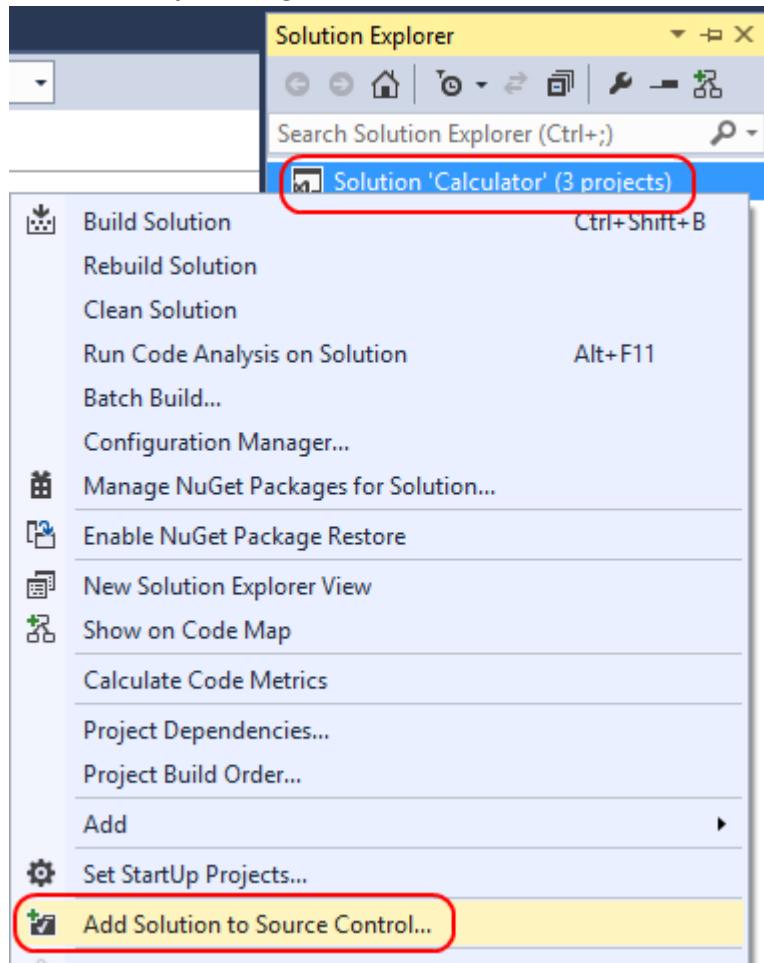


16. From the main menu, select **File | Close Solution**.

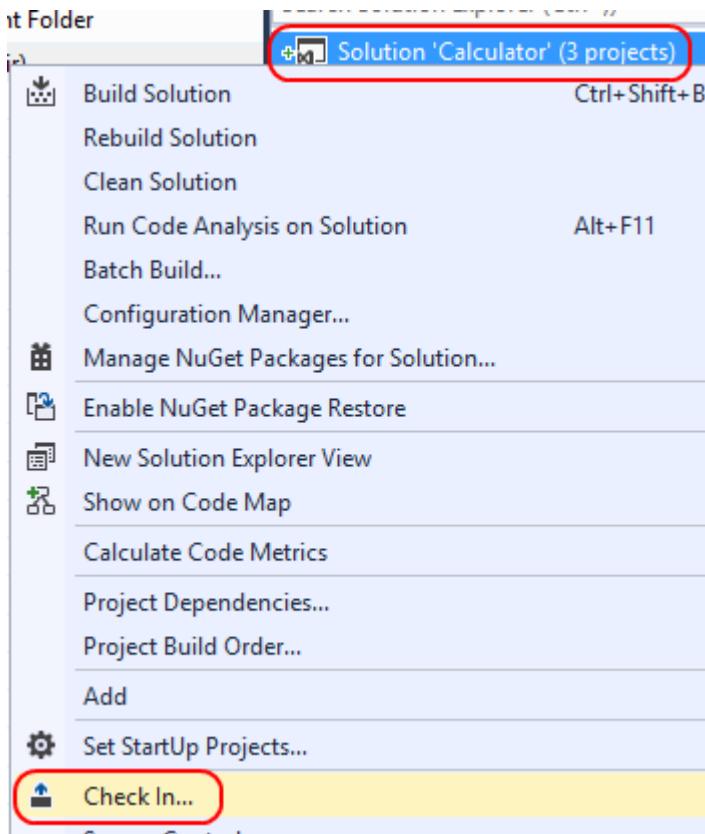
17. Using **Windows Explorer**, move the entire **Calculator Solution** folder into the folder you've mapped for source control. Note that the **Unit testing** team project will be in a **Unit testing** folder inside the map root, inside which there should only be the **BuildProcessTemplates** folder you just downloaded during the "get latest" step. The result should look something like the screenshot below.

Name	Date modified	Type
BuildProcessTemplates	1/29/2014 11:59 AM	File folder
Calculator Solution	1/28/2014 3:17 PM	File folder

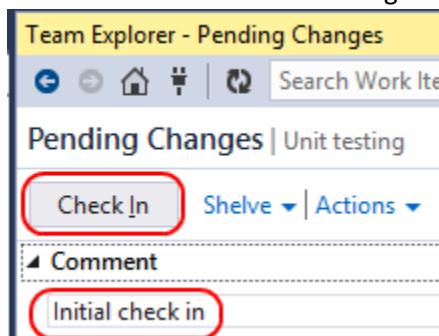
18. In **Visual Studio**, select **File | Open | Project/Solution...** and select the **Calculator.sln** file from the location you just moved it to.
19. In **Solution Explorer**, right-click the solution node and select **Add Solution to Source Control....**



20. Again, right-click the solution node and select **Check In....**



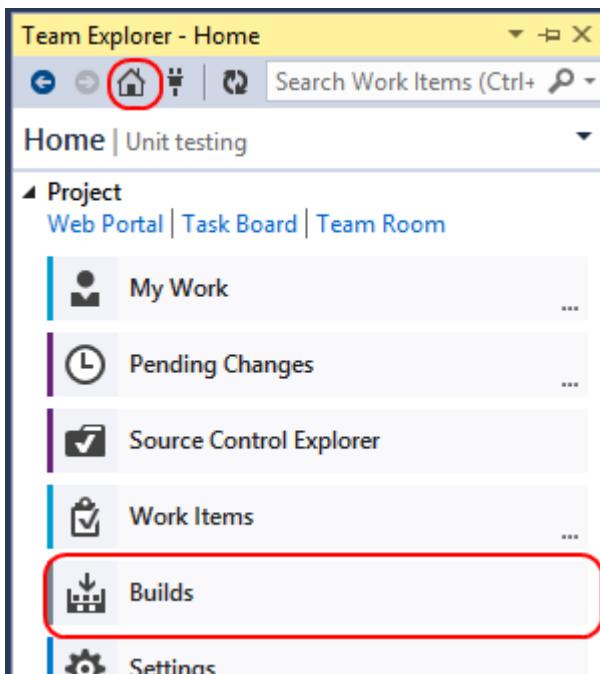
21. In **Team Explorer**, type “Initial check in” as the **Comment** and click **Check In**. If prompted to confirm, check the box not to be asked again and click **Yes**.



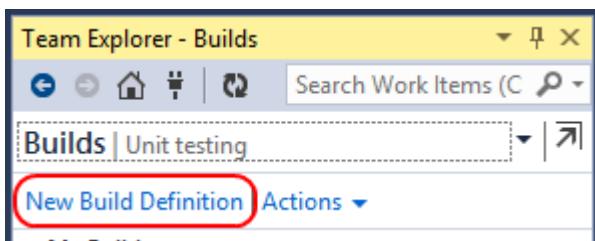
Task 2: Creating a build definition

In this task, you’ll create a build definition that instructs Visual Studio Online to build your project and run all available unit tests after a check in. Build definitions go well beyond just defining how the solution is built. They also perform additional tasks before, during, and after the build process, such as running unit tests.

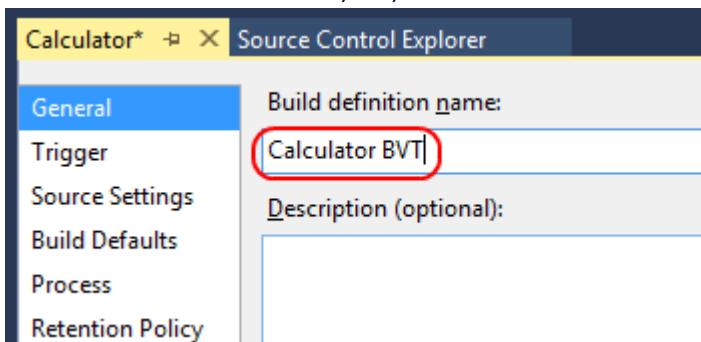
1. In **Team Explorer**, click the **Home** button and then click **Builds**.



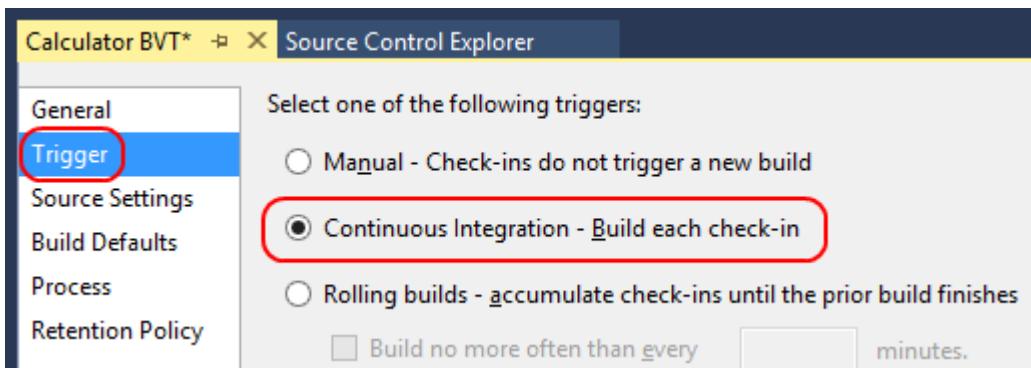
2. Click the **New Build Definition** link to create a new build definition.



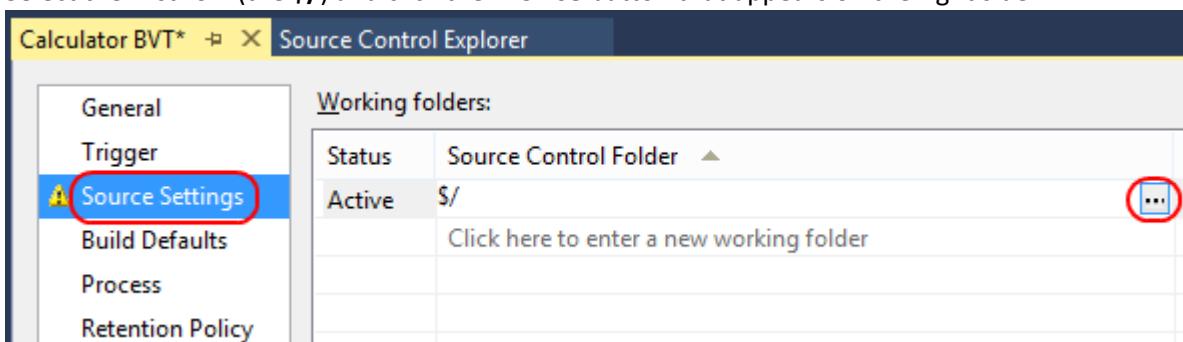
3. On the **General** page of the new build definition, type “Calculator BVT” as the **Build definition name**. “BVT” stands for “build verification test” and is a common term for automated builds that perform automated tests. Note that there is also the similarly named “BDT”, which refers to “build, deploy, & test”, which is a process that performs a deployment after the build, often including resources such as databases, etc, which are then tested.



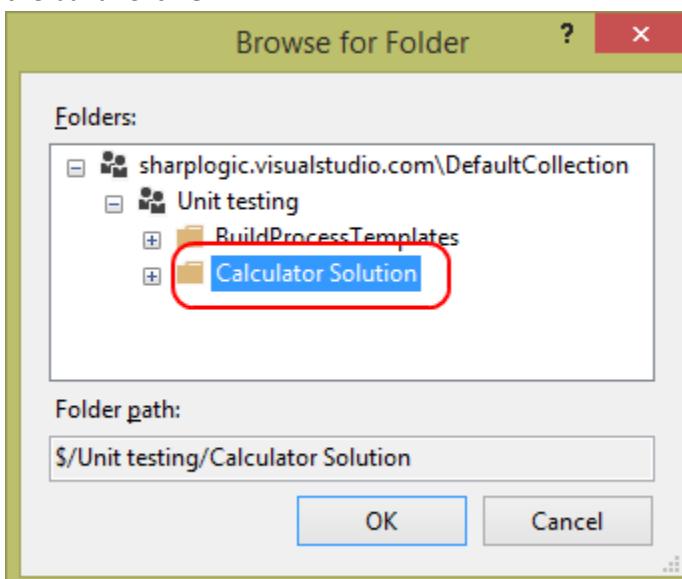
4. Select the **Trigger** tab and select the option for **Continuous Integration**. This will ensure that this build is run every time a check-in occurs. Note that you can also manually force (or “queue”) any build.



5. Select the **Source Settings** tab. On this page, you can configure the working folders needed to properly run the build. You should always refine this to the minimal source control folders required. Select the first row (the \$/) and click the **Browse** button that appears on the right side.



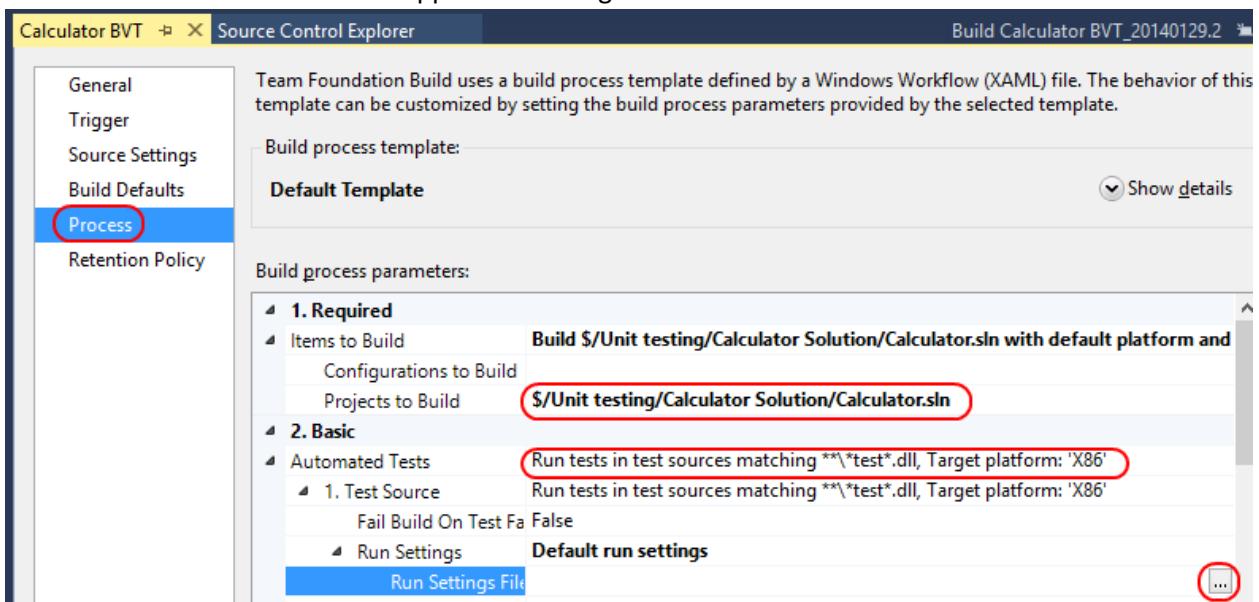
6. Navigate down to select the **Unit Testing | Calculator Solution** folder. This is all that's needed for the build. Click **OK**.



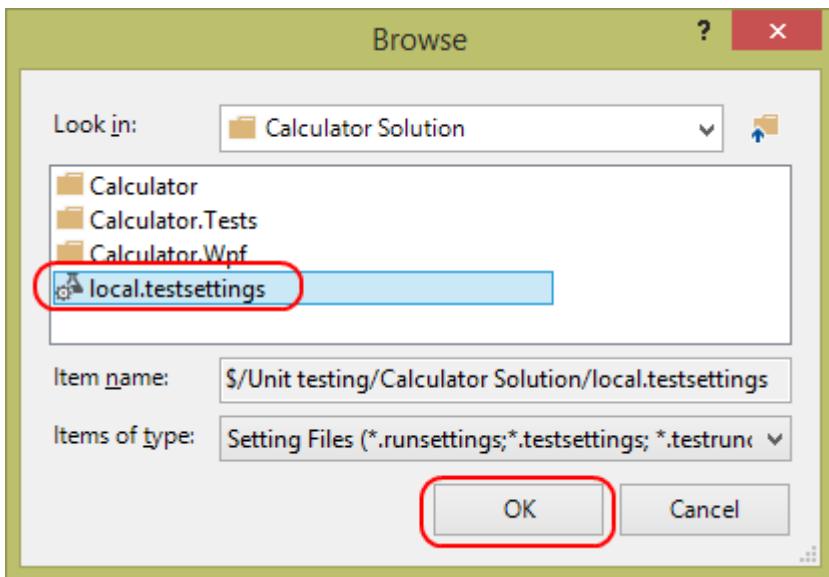
7. Select the **Build Defaults** tab. If running automated builds, you'll need to specify the controller. Since this will be run in the cloud using **Visual Studio Online**, select **Hosted Build Controller** as the **Build controller**.



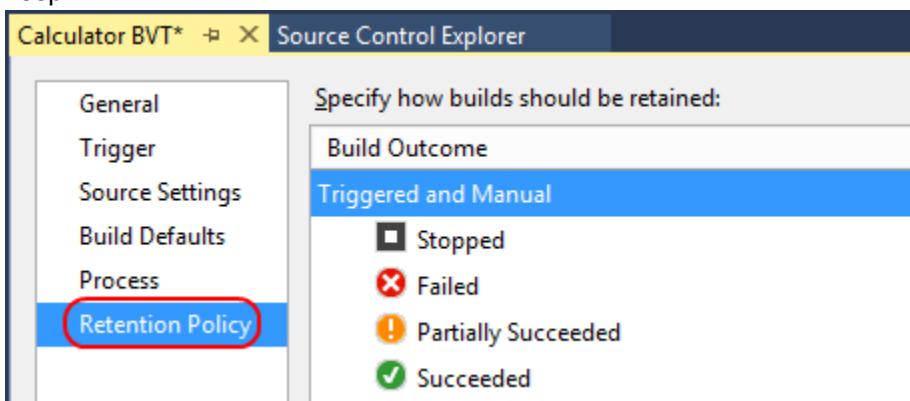
8. Select the **Process** tab. This page provides the full configurability for what actually happens in the build process. The default build definition is very useful, and will try to infer what to build (such as when there's only one solution) as well as default to automatically using any assembly with the term "**test**" in it for running automated tests. You can configure all of this, but it works for the purposes of this lab. One additional piece of configuration is to set the **Run Settings File**, so select the field and click the **Browse** button that appears at the right.



9. Select the **local.testsettings** file and click **OK**.



10. Select **Retention Policy**. This is the final configuration page and allows you specify which builds to keep.

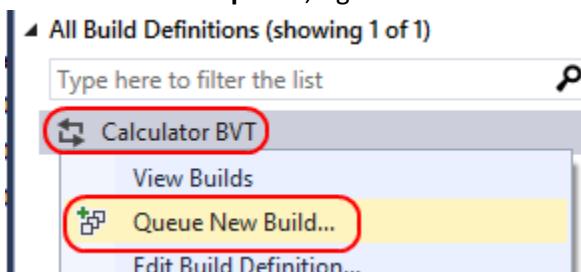


11. Press **Ctrl+S** to save the build definition.

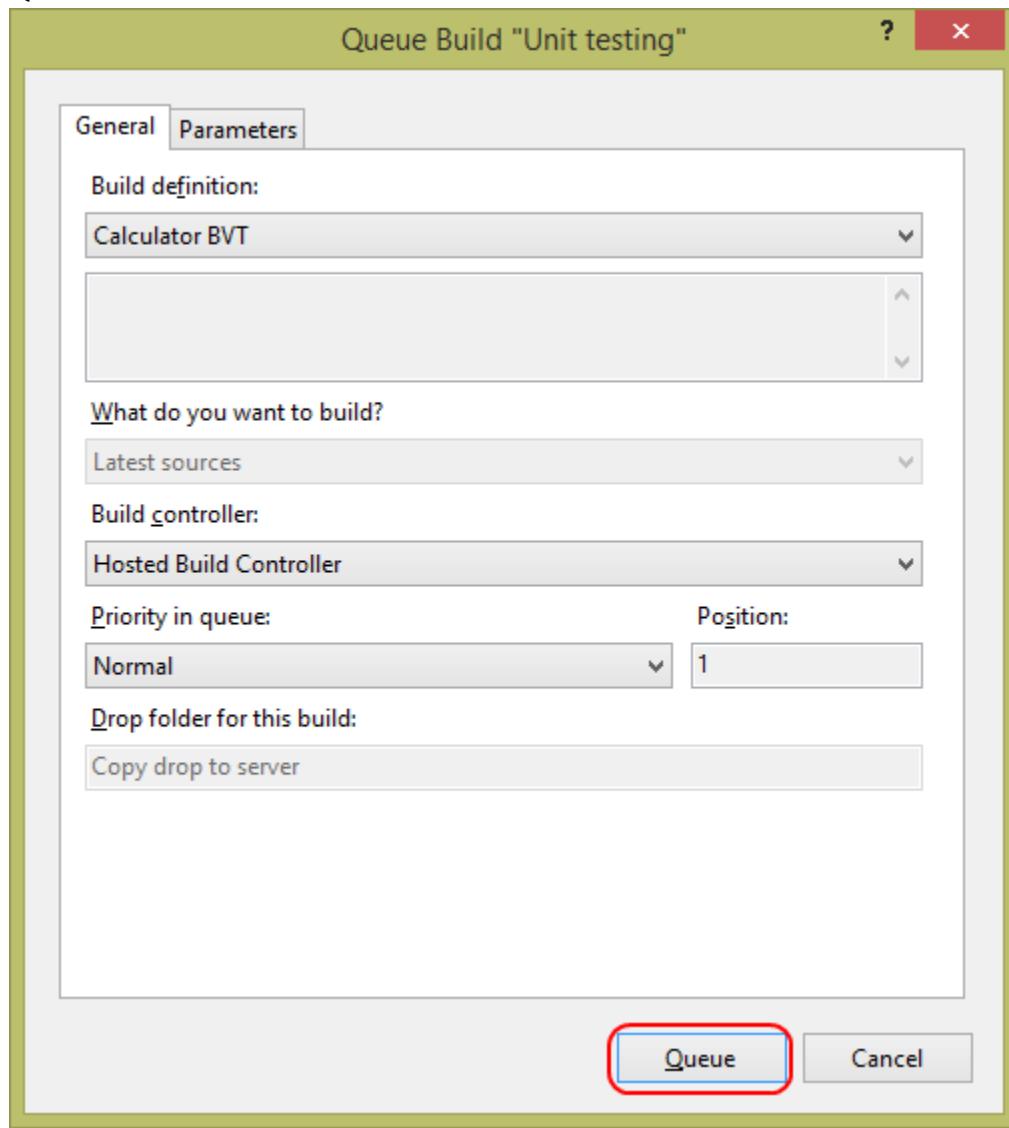
Task 3: Running builds and tests on the server

In this task, you'll use the build definition created in the last task to run builds that also execute the provided unit tests.

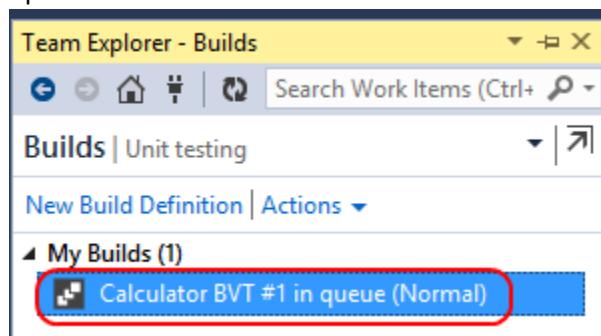
1. To test the build definition, you can manually queue a build for the build service. In the **Builds** section of **Team Explorer**, right-click the build and select **Queue New Build...**.



2. In the **Queue Build** dialog, you can customize exactly how the build will be set up. Since you already set the default **Build controller** during the creation of the build definition, you can just click the **Queue** button.



3. After the build has been queued, it'll appear in **Team Explorer**. You can double-click the build to open the build overview.



Depending on how much build traffic you're competing with today, it may take a few minutes before your build begins.

Build Request 310 – Queued

[View Queue](#) | [View Build Details](#) | [Cancel](#)

Requested by Ed Kaim for Manual using Calculator BVT (Unit testing)
Queued 6 seconds ago (Calculator BVT)

Queue Details

1 requests in the queue for this controller

1 requests in the queue for this definition

Position in the queue is 1

4. Scroll down to the **Build Summary** section and click the link to your build. It should start with "Calculator BVT".

Build Summary

► [Calculator BVT_20140129.3](#), started 42 seconds ago, currently in progress

Includes the following requests:

[Build Request 310](#), requested by Ed Kaim 87 seconds ago, In Progress

This will bring you to a detailed view of the build. The page should auto-refresh every 30 seconds. When the build finishes, you should see a page similar to this one.

Calculator BVT

Source Control Explorer

Calculator BVT_20140129.3 - Build succeeded

[View Summary](#) | [View Log - Open Drop Folder](#) | [Diagnostics](#) ▾ | [<No Quality Assigned>](#) ▾ | [Actions](#) ▾

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 344
Ran for 2.8 minutes (Hosted Build Controller), completed 0 seconds ago

5. Scroll down to the test run details. This outlines the results of your unit tests. Click the link to open the test results.

► 1 test run completed - 100% pass rate

[buildguest@BUILD-0048 2014-01-29 20:45:45 Any CPU Debug](#) 10 of 10 test(s) passed

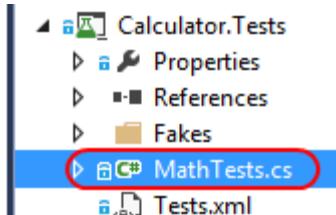
No Code Coverage Results

The test results will open in Visual Studio and provide the same details you would get if they ran locally.

Test Results				
	Result	Test Name	ID	Error Message
<input type="checkbox"/>	Passed	AddSimple	Calculator.Tests.MathTests.AddSimple	
<input type="checkbox"/>	Passed	AddSimple2	Calculator.Tests.MathTests.AddSimple2	
<input type="checkbox"/>	Passed	MultipleOperations	Calculator.Tests.MathTests.MultipleOperations	
<input type="checkbox"/>	Passed	AddDataDriven	Calculator.Tests.MathTests.AddDataDriven	
<input type="checkbox"/>	Passed	AddDataDriven (Data Row 0)	Calculator.Tests.MathTests.AddDataDriven (Data Row 0)	
<input type="checkbox"/>	Passed	AddDataDriven (Data Row 1)	Calculator.Tests.MathTests.AddDataDriven (Data Row 1)	
<input type="checkbox"/>	Passed	AddDataDriven (Data Row 2)	Calculator.Tests.MathTests.AddDataDriven (Data Row 2)	
<input type="checkbox"/>	Passed	Back	Calculator.Tests.MathTests.Back	
<input type="checkbox"/>	Passed	RandomSimple	Calculator.Tests.MathTests.RandomSimple	
<input type="checkbox"/>	Passed	AddSecondsSimple	Calculator.Tests.MathTests.AddSecondsSimple	

Now that you've confirmed that the build works properly, it's time to check in a change to confirm that it kicks off automatically.

6. Open **MathTests.cs** if not already open.

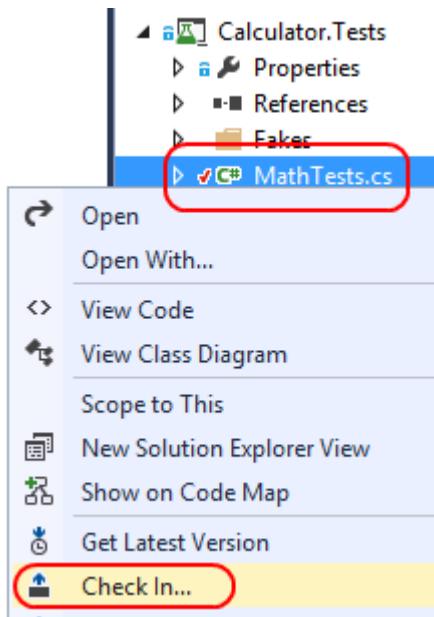


7. Add the following test. It is designed to fail.

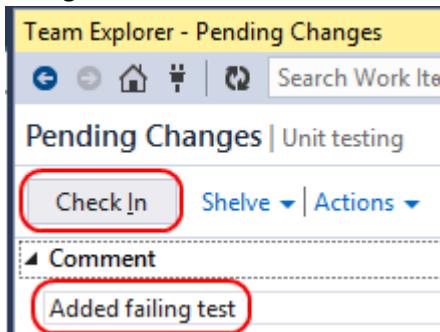
```
C#
[TestMethod]
public void Fail()
{
    calculator.CurrentValue = 1;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 1;
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(0, calculator.CurrentValue);
}
```

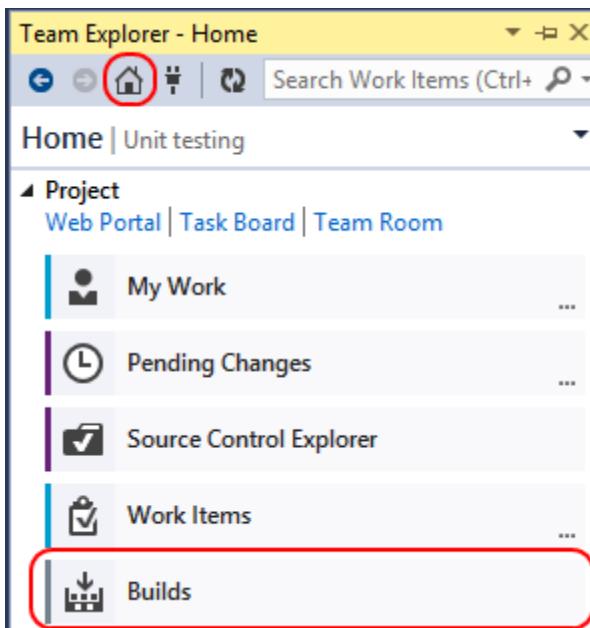
8. Press **Ctrl+S** to save the file.
9. In **Solution Explorer**, right-click **MathTests.cs** and select **Check In...**



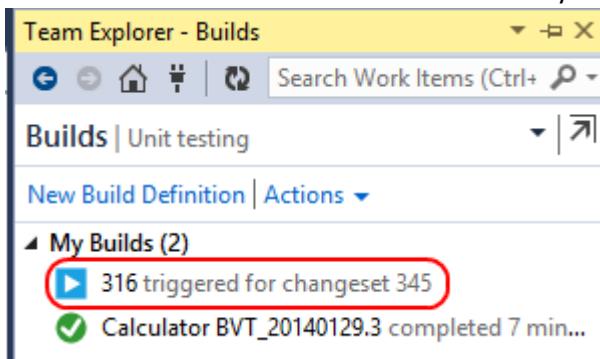
10. In **Team Explorer**, type “**Added failing test**” as the **Comment** and click **Check In** to check in the change.



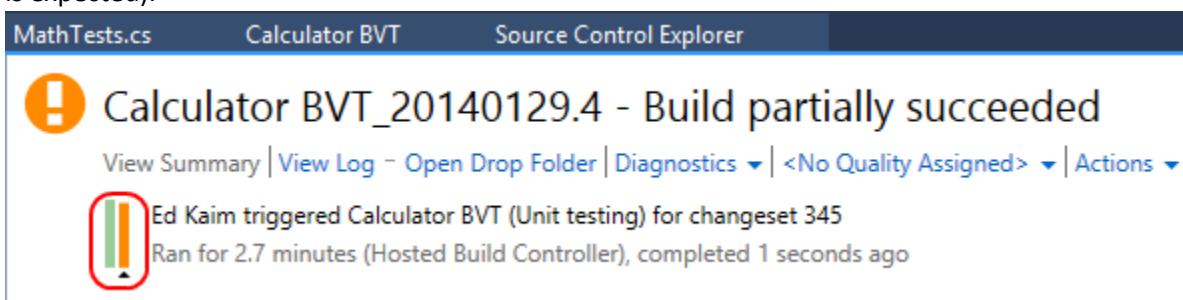
11. Click the **Home** button, followed by the **Builds** button.



12. There should now be a new build automatically triggered from the check in. Double-click it to open.



13. Click through to the build summary. When it finishes, you should get a “Build partially succeeded” message. In this case, it means that the build was successful, but that not all the tests passed (which is expected).



14. Scroll down to find and expand the test results. You'll see that the expected test failed.

```

▲ ✘ 1 test run completed - 90% pass rate
  ▲ ✘ buildguest@BUILD-0063 2014-01-29 20:55:22_Any CPU_Debug, 10 of 11 test(s) passed
    showing 1 of 1 failure(s)
      ▲ Fail failed.

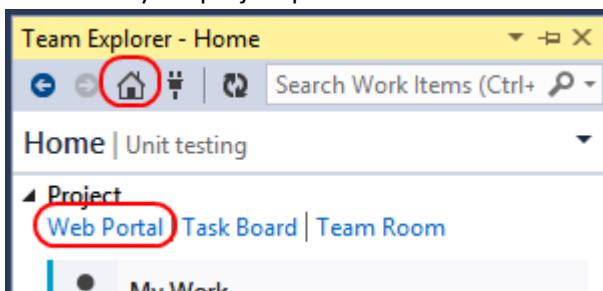
        Assert.AreEqual failed. Expected:<0>, Actual:<2>.
        at Calculator.Tests.MathTests.Fail() in c:\a\src\Calculator.Tests\MathTests.cs:line 131

```

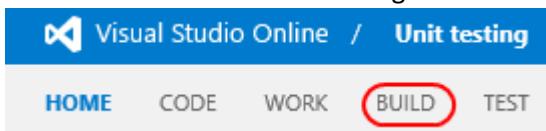
Task 4: Viewing and queuing builds on Visual Studio Online

In this task, you'll explore the build and test support provided by the Visual Studio Online portal.

1. In **Team Explorer**, click the **Home** button and then click the **Web Portal** link. This will open a browser window to your project portal.



2. Click the **Build** link from the navigation menu.



3. By default, this view will provide you with a list of today's completed builds. You can change the filter in the top right corner to customize what builds are shown. You can also queue a build from here, which would present you with similar options to the ones from Visual Studio. Double-click the first build, which is the most recent one.

All build definitions

Queued **Completed** Deployed

Queue build...

Name	Build Definition
Calculator BVT_20140129.4	Calculator BVT
Calculator BVT_20140129.3	Calculator BVT

The build results appear in a way that's similar to what you saw in Visual Studio. You also have the ability to download the whole drop as a zip file, mark the build to be retained indefinitely, or even delete the build.

▼ Calculator BVT_20140129.4 - Build partially succeeded

[Summary](#) [Log](#) [Diagnostics](#)

[Download drop as zip](#) | [Retain indefinitely](#) |  | <No quality assigned> ▾

|| Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345

----- Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

Latest activity

Build last modified by Elastic Build (sharplogic) 3.9 minutes ago.

Associated changesets

[Changeset 345](#) Checked in by Ed Kaim

4. Click the **Log** link. The log view provides the steps followed during the build workflow, along with their warnings and errors, if applicable.

▼ Calculator BVT_20140129.4 - Build partially succeeded

[Summary](#) [Log](#) [Diagnostics](#)

[Download drop as zip](#) | [Retain indefinitely](#) |  | <No quality assigned> ▾

|| Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345

----- Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

Overall Build Process

Update Build Number

Run On Agent (reserved build agent Hosted Build Agent)

5. Click the **Diagnostics** link. This view is similar to the log view, except that it offers a verbose level of detail regarding each individual action taken and its result. This view is extremely helpful when troubleshooting build definitions.

➡ Calculator BVT_20140129.4 - Build partially succeeded

Summary Log **Diagnostics**

Download drop as zip | Retain indefinitely |  | <No quality assigned> ▾

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

View logs ▾ | Hide properties | Show properties | Next error | Previous error

Overall Build Process ▾

Get the Build
Update Drop Location

- Finally, you can assign a quality level to this build. Since it failed, select **Rejected** so that other team members can know to avoid it for deployment.

➡ Calculator BVT_20140129.4 - Build partially succeeded

Summary Log **Diagnostics** <No quality assigned>

Download drop as zip | Retain indefinitely |  | <No quality assigned> ▾

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

View logs ▾ | Hide properties | Show properties

Overall Build Process ▾

Get the Build
Update Drop Location
Update Build Number ▾
If Build Reason is Triggered ▾
Set Drop Location ▾
If Build Reason is ValidateShelveset ▾

<No quality assigned> ▾

Initial Test Passed
Lab Test Passed
Ready for Deployment
Ready for Initial Test
Rejected

- Besides tracking and viewing builds, you can also queue new builds from the portal. In the left panel, locate the **Build Definitions** node. Right-click **Calculator BVT** and select **Queue build....**

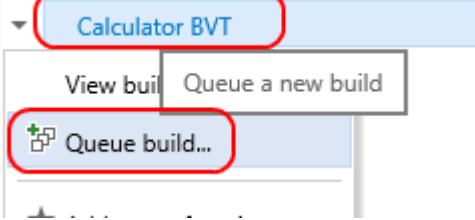
No favorite build definition found.

▲ Team favorites

No team build definition found.

▲ Build definitions

All build definitions

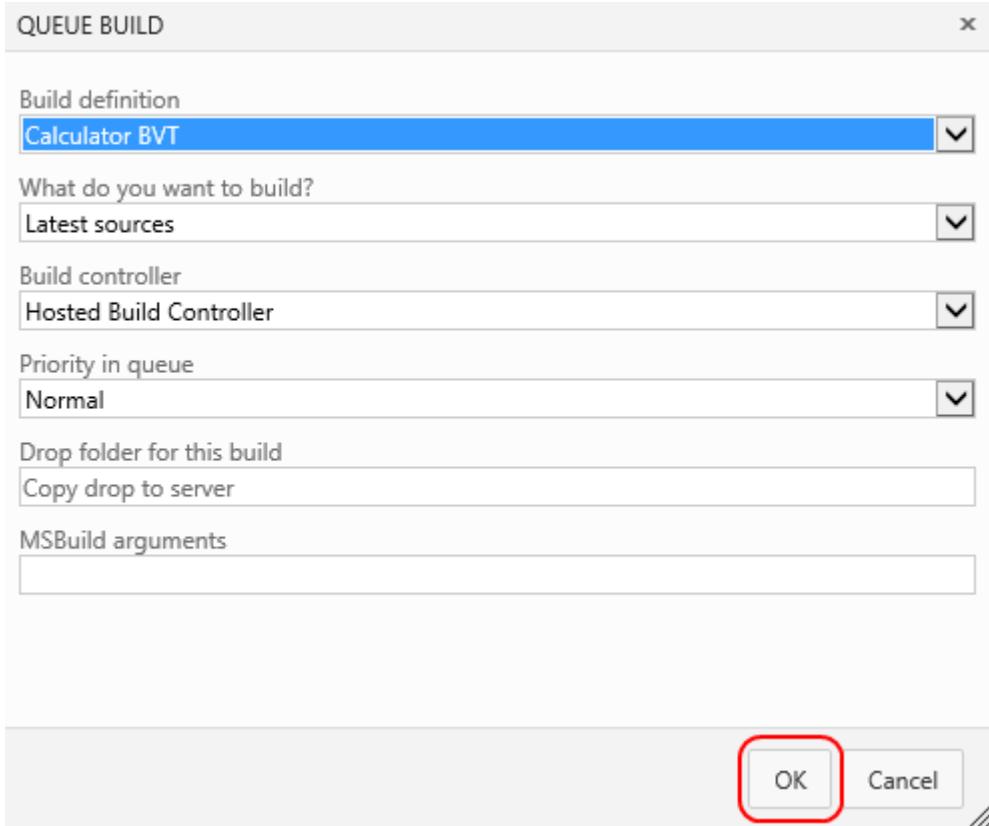


Calculator BVT

View build Queue a new build

+ Queue build...

8. In the **Queue Build** dialog, accept the defaults and click **OK**. This will queue a build just like you did earlier from Visual Studio. Note that virtually all of the build queuing and tracking functionality is shared between Visual Studio and the Visual Studio Online project portal.



9. After queuing, locate the new build in the main view and double-click it.

Queued Completed Deployed

Queue build...

	Id	Definition	Priority	Queued
	313	Calculator BVT	Normal	3 minutes ago
	314	Calculator RVT	Normal	less than a mi

10. Note that you can view build progress in the portal just like in Visual Studio.

Building Calculator BVT_20140212.2

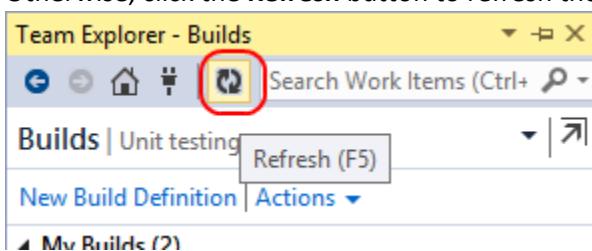
Summary **Log**

Open drop folder |

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 351
Running for 2.3 minutes (Hosted Build Controller)

- Overall Build Process
 - Update Build Number
 - Run On Agent (reserved build agent Hosted Build Agent)
 - Create Workspace
 - Get Workspace
 - Create Label
 - Compile, Test, and Associate Changesets and Work Items
 - Compile and Test
 - Run MSBuild for Project

11. Back in **Visual Studio**, switch to the **Builds** section of **Team Explorer** if it's not already open. Otherwise, click the **Refresh** button to refresh the view.



12. The new build will appear at the top. Double-click it to open.

Team Explorer - Builds

Builds | Unit testing

New Build Definition | Actions ▾

▲ My Builds (3)

▶ Calculator BVT_20140212.3 started 6 second...

Now you're viewing the same experience you just saw in the portal. In many cases you may find yourself using Visual Studio, but it's great to know that so much of the development experience is still available, even when working from a phone or tablet browser.

▶ Building Calculator BVT_20140212.3

View Summary | View Log | Diagnostics ▾ | Actions ▾

|| Ed Kaim triggered Calculator BVT (Unit testing) for changeset 351
Running for 36 seconds (Hosted Build Controller)

— Activity Log | Next Error | Next Warning | Auto Refresh: On —————

▶ Overall Build Process

Update Build Number

▶ Run On Agent (reserved build agent Hosted Build Agent)

Dependency Injection

What is Dependency Injection Design Pattern in C#?

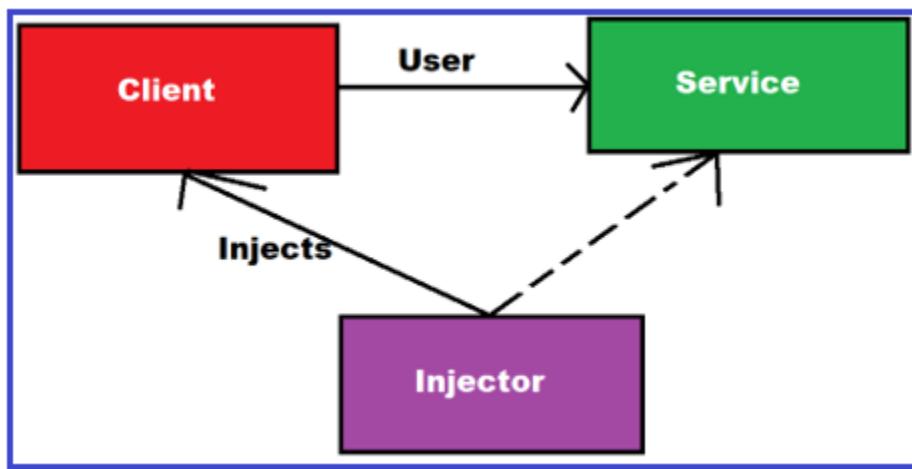
The Dependency Injection Design Pattern in C# is a process in which we are injecting the object of a class into a class that depends on that object. The Dependency Injection design pattern is the most commonly used design pattern nowadays to remove the dependencies between the objects.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependency objects outside of a class and provides those objects to a class in different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

Dependency Injection pattern involves 3 types of classes:

1. **Client Class:** The Client class (dependent class) is a class that depends on the service class.
2. **Service Class:** The Service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The Injector class injects the service class object into the client class.

For better understanding, please have a look at the following diagram.



As you can see above in the above diagram, the injector class creates an object of the service class and injects that object to a client class. In this way, the Dependency Injection pattern separates the responsibility of creating an object of the service class out of the client class.

Different Types of Dependency Injection in C#?

The injector class injects the dependency object to a class in three different ways. They are as follows.

Constructor Injection: When the Injector injects the dependency object (i.e. service) through the client class constructor, then it is called as Constructor Injection.

Property Injection: When the Injector injects the dependency object (i.e. service) through the public property of the client class, then it is called as Property Injection. This is also called as the Setter Injection.

Method Injection: When the Injector injects the dependency object (i.e. service) through a public method of the client class, then it is called as Method Injection. In this case, the client class implements an interface that declares the method(s) to supply the dependency object and the injector uses this interface to supply the dependency object (i.e. service) to the client class.

Now create 3 classes **Employee.cs**, **EmployeeDAL.cs** and **EmployeeBL.cs** as shown below

Employee.cs

```
namespace DependencyInjectionExample
{
    public class Employee
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Department { get; set; }
    }
}
```

EmployeeDAL.cs

```
namespace DependencyInjectionExample
{
    public class EmployeeDAL
    {
        public List<Employee> SelectAllEmployees()
        {
            List<Employee> ListEmployees = new List<Employee>();
            //Get the Employees from the Database
            //for now we are hard coded the employees
            ListEmployees.Add(new Employee() { ID = 1, Name = "Pranaya", Department = "IT" });
            ListEmployees.Add(new Employee() { ID = 2, Name = "Kumar", Department = "HR" });
            ListEmployees.Add(new Employee() { ID = 3, Name = "Rout", Department = "Payroll" });
            return ListEmployees;
        }
    }
}
```

EmployeeBL.cs

```
namespace DependencyInjectionExample
{
    public class EmployeeBL
```

```

{
public EmployeeDAL employeeDAL;

public List<Employee> GetAllEmployees()
{
employeeDAL = new EmployeeDAL();
return employeeDAL.SelectAllEmployees();
}
}
}

```

In the above example, in order to get the data, the **EmployeeBL** class depends on the **EmployeeDAL** class. In the **GetAllEmployees()** method of the **EmployeeBL** class, we create an instance of the **EmployeeDAL** (Employee Data Access Layer) class and then invoke the **SelectAllEmployees()** method. This is tight coupling because the **EmployeeDAL** is tightly coupled with the **EmployeeBL** class. Every time the **EmployeeDAL** class changes, the **EmployeeBL** class also needs to change.

Constructor Injection

Let us see how to use the constructor injection to make these classes loosely coupled.

Modify the EmployeeDAL.cs file as shown below

```

namespace DependencyInjectionExample
{
public interface IEmployeeDAL
{
List<Employee> SelectAllEmployees();
}

public class EmployeeDAL : IEmployeeDAL
{
public List<Employee> SelectAllEmployees()
{
List<Employee> ListEmployees = new List<Employee>();
//Get the Employees from the Database
//for now we are hard coded the employees
ListEmployees.Add(new Employee() { ID = 1, Name = "Pranaya", Department = "IT" });
ListEmployees.Add(new Employee() { ID = 2, Name = "Kumar", Department = "HR" });
ListEmployees.Add(new Employee() { ID = 3, Name = "Rout", Department = "Payroll" });
return ListEmployees;
}
}
}

```

As you can see, first we create one interface i.e IEmployeeDAL with the one method. Then that interface is implemented by the EmployeeDAL class. So the point that I need to keep focus is when you are going to use the dependency injection design pattern in c#, then the dependency object should be interface based. In our example, the EmployeeDAL is the

dependency object as this object is going to be used by the EmployeeBL class. So we created the interface and then implement that interface.

Modify the EmployeeBL.cs file as shown below

```
namespace DependencyInjectionExample
{
    public class EmployeeBL
    {
        public IEmployeeDAL employeeDAL;
        public EmployeeBL(IEmployeeDAL employeeDAL)
        {
            this.employeeDAL = employeeDAL;
        }
        public List<Employee> GetAllEmployees()
        {
            Return employeeDAL.SelectAllEmployees();
        }
    }
}
```

In the above example, we created one constructor which accepts one parameter of the dependency object type. The point that you need to keep focus is, the parameter of the constructor is of the type interface, not the concrete class. Now, this parameter can accept any concrete class object which implements this interface.

So here in the EmployeeBL class, we are not creating the object of the EmployeeDAL class. Instead, we are passing it as a parameter to the constructor of the EmployeeBL class. As we are injecting the dependency object through the constructor, it is called as constructor dependency injection in C#.

Use EmployeeBL class in our Main method of Program class:

```
namespace DependencyInjectionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            EmployeeBL employeeBL = new EmployeeBL(new EmployeeDAL());
            List<Employee> ListEmployee = employeeBL.GetAllEmployees();
            foreach(Employee emp in ListEmployee)
            {

```

```

Console.WriteLine("ID = {0}, Name = {1}, Department = {2}", emp.ID, emp.Name,
emp.Department);
}
Console.ReadKey();
}
}
}

```

Now run the application and you will see the output as expected as shown below.

```

ID = 1, Name = Pranaya, Department = IT
ID = 2, Name = Kumar, Department = HR
ID = 3, Name = Rout, Department = Payroll

```

Advantages of Constructor Dependency Injection

1. The Constructor Dependency Injection Design Pattern makes a strong dependency contract
2. This design pattern support testing as the dependencies are passed through the constructor.

Property and Method Dependency Injection in C#

In Property Dependency Injection, we need to supply the dependency object through a public property of the client class. Let us see an example to understand how we can implement the Property or you can say setter dependency injection in C#.

Modify the EmployeeBL class as shown below

```

namespace DependencyInjectionExample
{
public class EmployeeBL
{
private IEmployeeDAL employeeDAL;
public IEmployeeDAL employeeDataObject
{
set
{
this.employeeDAL = value;
}
get
{
if (employeeDataObject == null)
{
throw new Exception("Employee is not initialized");
}
}
}

```

```

else
{
return employeeDAL;
}
}
}

public List<Employee> GetAllEmployees()
{
return employeeDAL.SelectAllEmployees();
}
}
}
}

```

As you can see in the above example, we are injecting the dependency object through a public property of the EmployeeBL class. As we are setting the object through the setter property, we can call this as Setter Dependency Injection in C#. Here we need to use the property EmployeeDataObject in order to access the instance of IEmployeeDAL.

Change the Main method of Program class as shown below to inject the object through a property.

```

namespace DependencyInjectionExample
{
class Program
{
static void Main(string[] args)
{
EmployeeBL employeeBL = new EmployeeBL();
employeeBL.employeeDataObject = new EmployeeDAL();
List<Employee> ListEmployee = employeeBL.GetAllEmployees();
foreach(Employee emp in ListEmployee)
{
Console.WriteLine("ID = {0}, Name = {1}, Department = {2}", emp.ID, emp.Name,
emp.Department);
}
Console.ReadKey();
}
}
}

```

Now run the application and you will see the output as expected as shown below.

```
ID = 1, Name = Pranaya, Department = IT  
ID = 2, Name = Kumar, Department = HR  
ID = 3, Name = Rout, Department = Payroll
```

The Property or Setter Dependency Injection in C# does not require the constructor to be changed. Here the dependency objects are going to be passed through the public properties of the client class. We need to use the Setter or Property Dependency Injection when we want to create the dependency object as late as possible or we can say when it is required.

When to use Property Dependency Injection over Constructor Injection and vice versa?

The Constructor Dependency Injection in C# is the standard for dependency injection. It ensures that all the dependency objects are initialized before we are going to invoke any methods or properties of the dependency object, as a result, it avoids the null reference exceptions.

The Setter/Property Dependency Injection in C# is rarely used in real-time applications. For example, if I have a class that has several methods but those methods do not depend on any other objects. Now we need to create a new method within the same class but that new method now depends on another object. If we use the constructor dependency injection here, then we need to change all the existing constructor calls where we created this class object. This can be a very difficult task if the project is a big one. Hence, in such scenarios, the Setter or Property Dependency Injection can be a good choice.

What is Method Dependency Injection in C#?

In Method Dependency Injection, we need to supply the dependency object through a public method of the client class. Let us see an example to understand how we can implement the Method dependency injection in C#.

Modify the EmployeeBL class as shown below

```
namespace DependencyInjectionExample  
{  
    public class EmployeeBL  
    {  
        public IEmployeeDAL employeeDAL;  
        public List<Employee> GetAllEmployees(IEmployeeDAL _employeeDAL)  
        {  
            employeeDAL = _employeeDAL;  
            return employeeDAL.SelectAllEmployees();  
        }  
    }  
}
```

Modify the Main method of Program class as shown below

```
namespace DependencyInjectionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create object of EmployeeBL class
            EmployeeBL employeeBL = new EmployeeBL();
            //Call to GetAllEmployees method with proper object.
            List<Employee> ListEmployee = employeeBL.GetAllEmployees(new EmployeeDAL());
            foreach (Employee emp in ListEmployee)
            {
                Console.WriteLine("ID = {0}, Name = {1}, Department = {2}", emp.ID, emp.Name,
                emp.Department);
            }
            Console.ReadKey();
        }
    }
}
```

Now run the application and see the output as expected as shown below

```
ID = 1, Name = Pranaya, Department = IT
ID = 2, Name = Kumar, Department = HR
ID = 3, Name = Rout, Department = Payroll
```

What are the advantages of using Dependency Injection in C#?

1. The Dependency Injection Design Pattern allows us to develop loosely coupled software components.
2. Using Dependency Injection, it is very easy to swap with a different implementation of a component, as long as the new component implements the interface type.

Dependency Injection Container:

There are a lot of Dependency Injection Containers available in the market which implements the dependency injection design pattern. Some of the commonly used Dependency Injection Containers are as follows.

1. **Unity**
2. **Castle Windsor**
3. **StructureMap**
4. **Spring.NET**