ASP.NET CORE

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

ASP.NET Core is the new web framework from Microsoft. ASP.NET Core is the framework you want to use for web development with .NET. At the end this tutorial, you will have everything you need to start using ASP.NET Core and write an application that can create, edit, and view data from a database.

Audience

This tutorial is designed for software programmers who would like to learn the basics of ASP.NET Core from scratch.

Prerequisites

You should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

Disclaimer & Copyright

Table of Contents

1. ASP.NET — Overview

ASP.NET Core is the new web framework from Microsoft. It has been redesigned from the ground up to be fast, flexible, modern, and work across different platforms. Moving forward, ASP.NET Core is the framework that can be used for web development with .NET. If you have any experience with MVC or Web API over the last few years, you will notice some familiar features. At the end this tutorial, you will have everything you need to start using ASP.NET Core and write an application that can create, edit, and view data from a database.

A Brief History of ASP.NET

ASP.NET has been used from many years to develop web applications. Since then, the framework went through a steady evolutionary change and finally led us to its most recent descendant ASP.NET Core 1.0.

- ASP.NET Core 1.0 is not a continuation of ASP.NET 4.6.
- It is a whole new framework, a side-by-side project which happily lives alongside everything else we know.
- It is an actual re-write of the current ASP.NET 4.6 framework, but much smaller and a lot more modular.
- Some people think that many things remain the same, but this is not entirely true. ASP.NET Core 1.0 is a big fundamental change to the ASP.NET landscape.

What is ASP.NET Core

ASP.NET Core is an open source and cloud-optimized web framework for developing modern web applications that can be developed and run on Windows, Linux and the Mac. It includes the MVC framework, which now combines the features of MVC and Web API into a single web programming framework.

- ASP.NET Core apps can run on .NET Core or on the full .NET Framework.
- It was architected to provide an optimized development framework for apps that are deployed to the cloud or run on-premises.
- It consists of modular components with minimal overhead, so you retain flexibility while constructing your solutions.
- You can develop and run your ASP.NET Core apps cross-platform on Windows, Mac and Linux.

Advantages of ASP.NET Core

ASP.NET Core comes with the following advantages:

- ASP.NET Core has a number of architectural changes that result in a much leaner and modular framework.
- ASP.NET Core is no longer based on System.Web.dll. It is based on a set of granular and well factored NuGet packages.
- This allows you to optimize your app to include just the NuGet packages you need.
- The benefits of a smaller app surface area include tighter security, reduced servicing, improved performance, and decreased costs

With ASP.NET Core, you can get the following improvements:

- Build and run cross-platform ASP.NET apps on Windows, Mac and Linux
- Built on .NET Core, which supports true side-by-side app versioning
- New tooling that simplifies modern wWeb development
- Single aligned web stack for Web UI and Web APIs
- Cloud-ready environment-based configuration
- Built-in support for dependency injection
- Tag Helpers which makes Razor markup more natural with HTML
- Ability to host on IIS or self-host in your own process

2. ASP.NET Core — Environment Setup

ASP.NET Core is a significant redesign of ASP.NET. This topic introduces the new concepts in ASP.NET Core and explains how they help you develop modern web apps.

To use ASP.NET Core in your application, the following must be installed in your system:

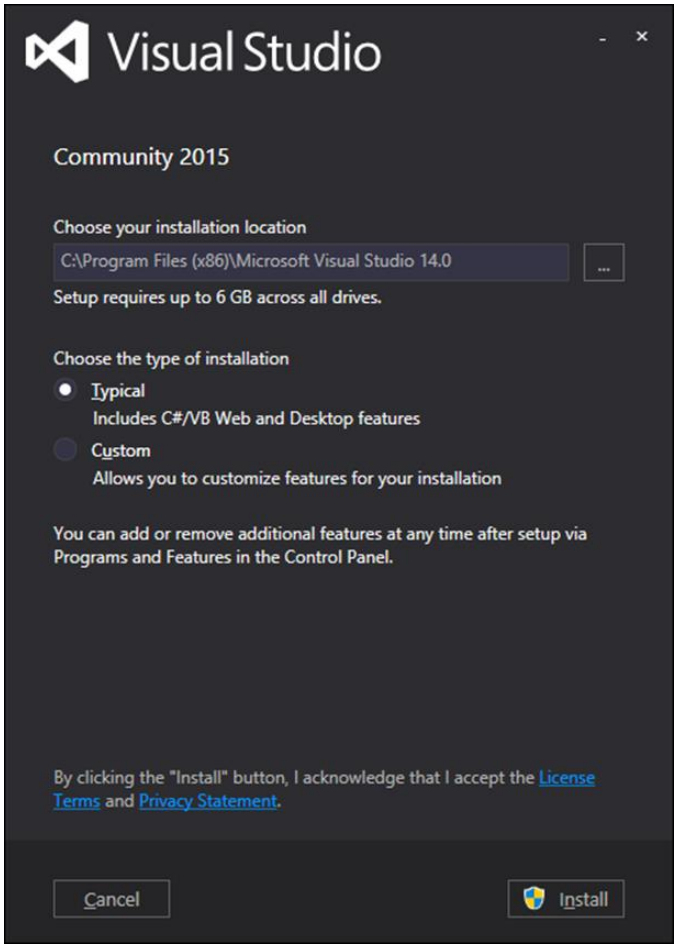
- Microsoft Visual Studio 2015
- Microsoft .NET Core 1.0.0 - VS 2015 Tooling Preview 2

Microsoft provides a free version of Visual Studio which also contains the SQL Server and it can be downloaded from https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx and Microsoft .NET Core 1.0.0 - VS 2015 Tooling Preview 2 can be downloaded from https://go.microsoft.com/fwlink/?LinkId=817245.

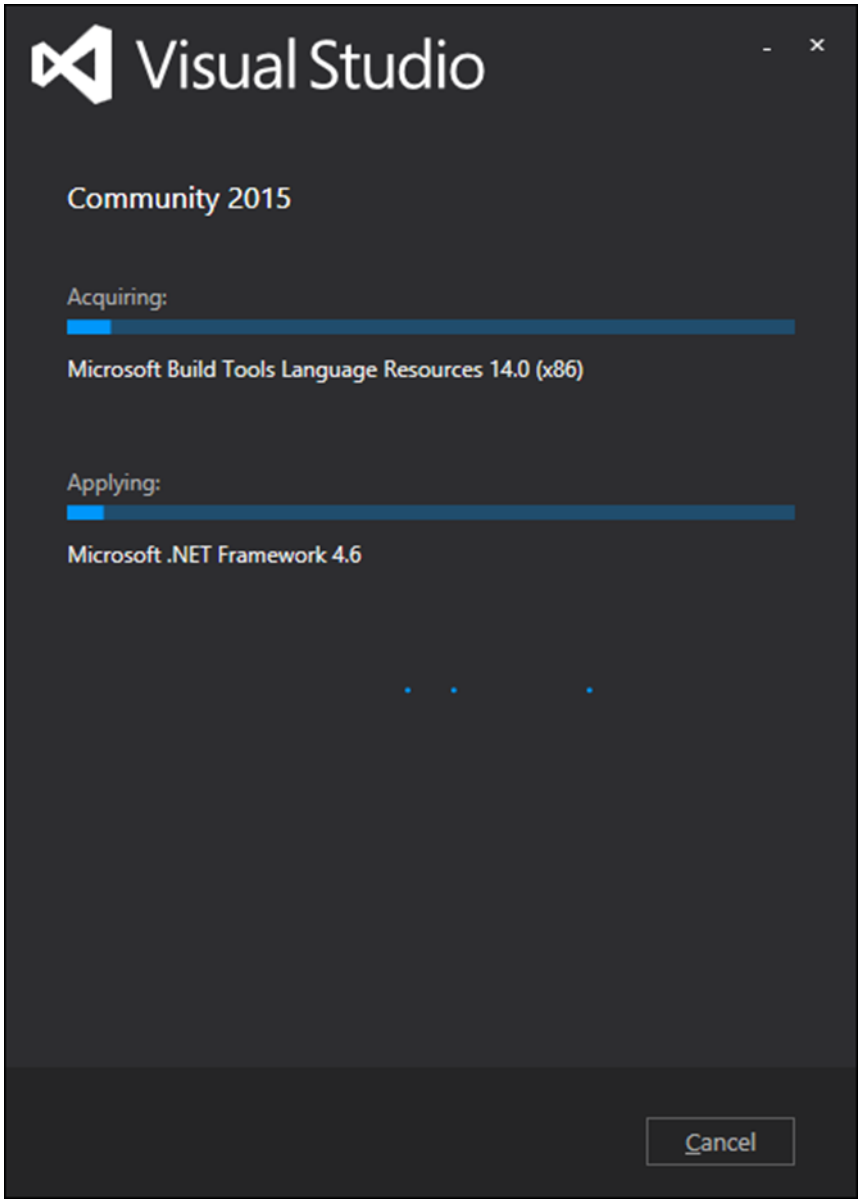
Installation of Microsoft Visual Studio 2015

Let us now understand the steps involved in the installation of

Step 1: Once downloading is completed, run the installer. The following dialog box will be displayed.



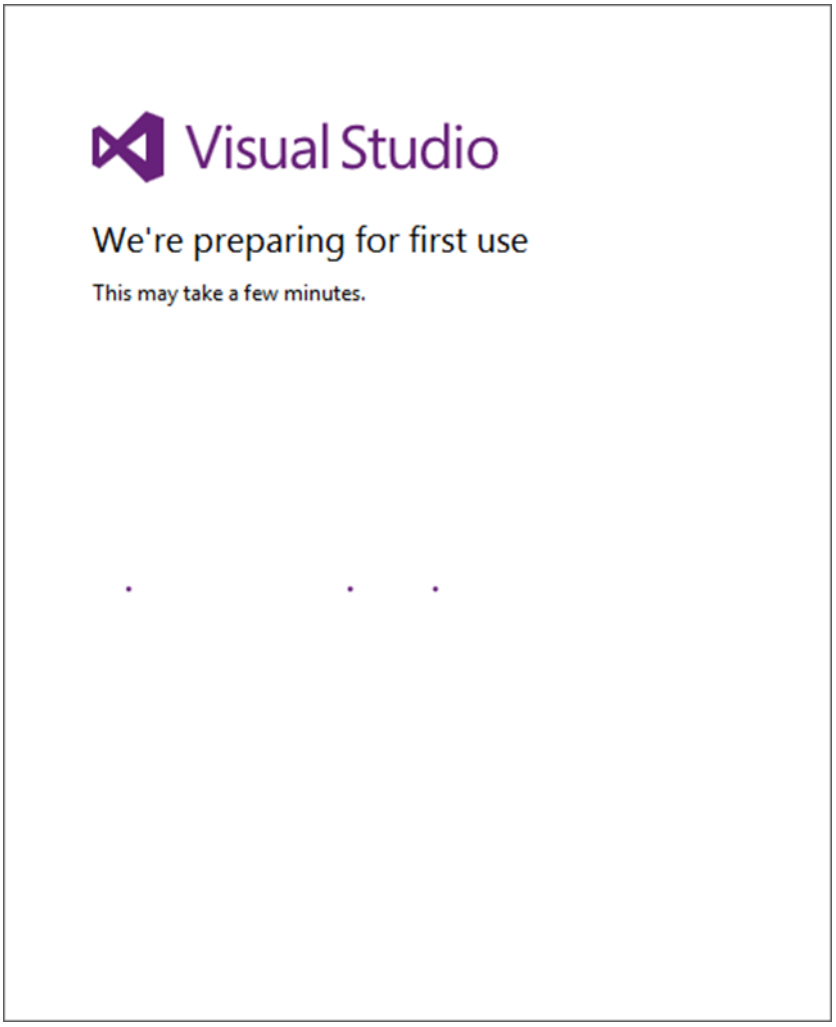
Step 2: Click the Install button as in the above screenshot. The installation process will then start.



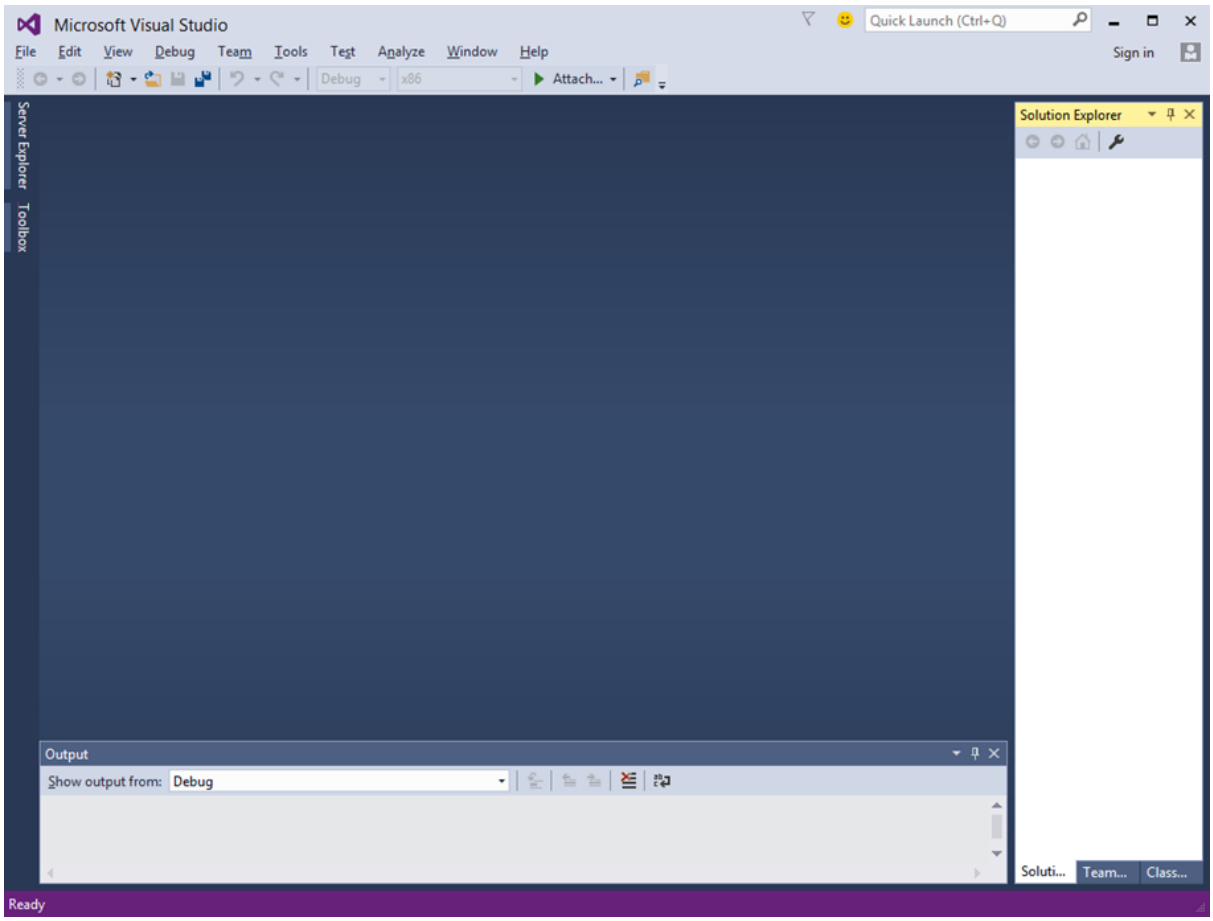
Step 3: Once the installation process is completed successfully, you will see the following dialog box.

Step 4: Close this dialog and restart your computer if required.

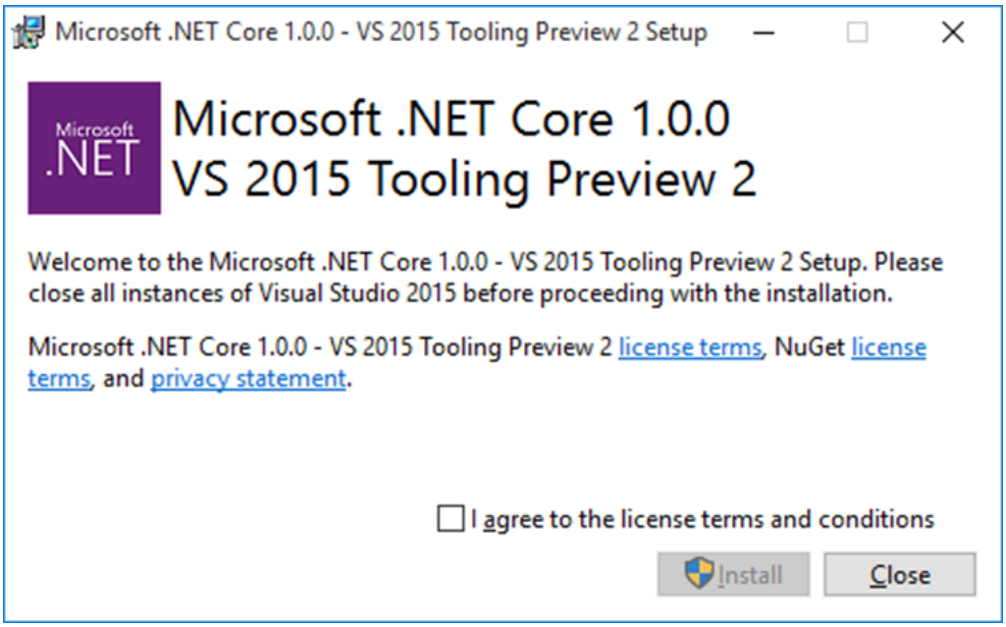
Step 5: Open the Visual studio from the Start menu. This will open the following dialog box and it will take some time for the first time (only for preparation).



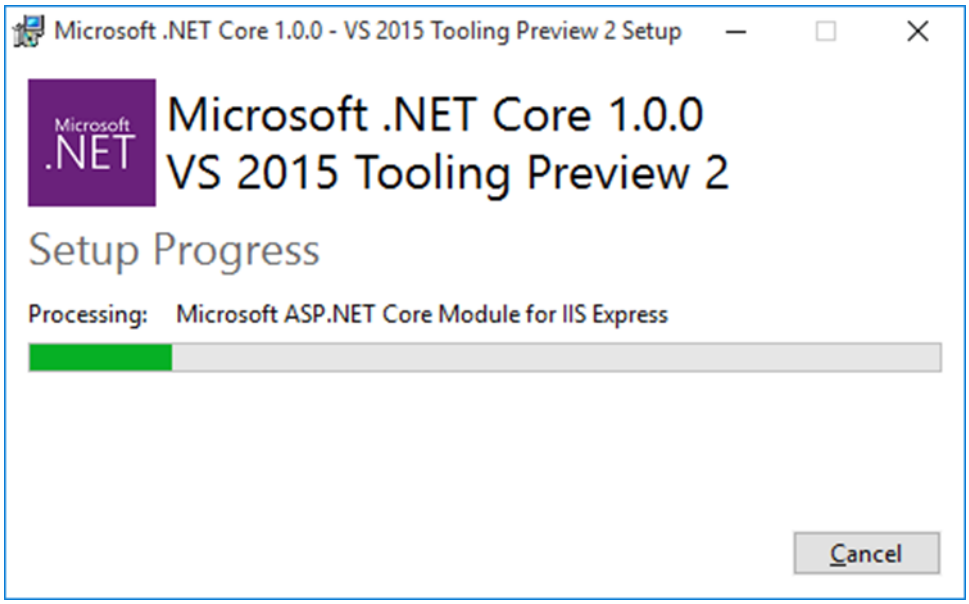
Step 6: You will now see the main window of the Visual studio.



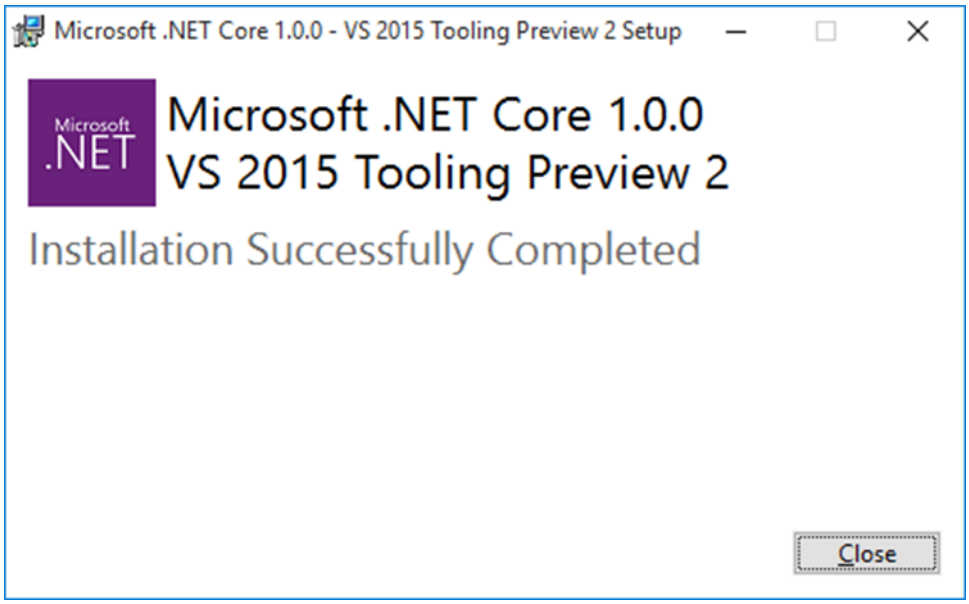
Step 7: Once the Visual Studio is installed, close the Visual Studio and launch the Microsoft .NET Core 1.0.0 - VS 2015 Tooling Preview 2



Step 8: Check the checkbox and click Install.



Step 9: Once the installation is completed, you will see the following message.

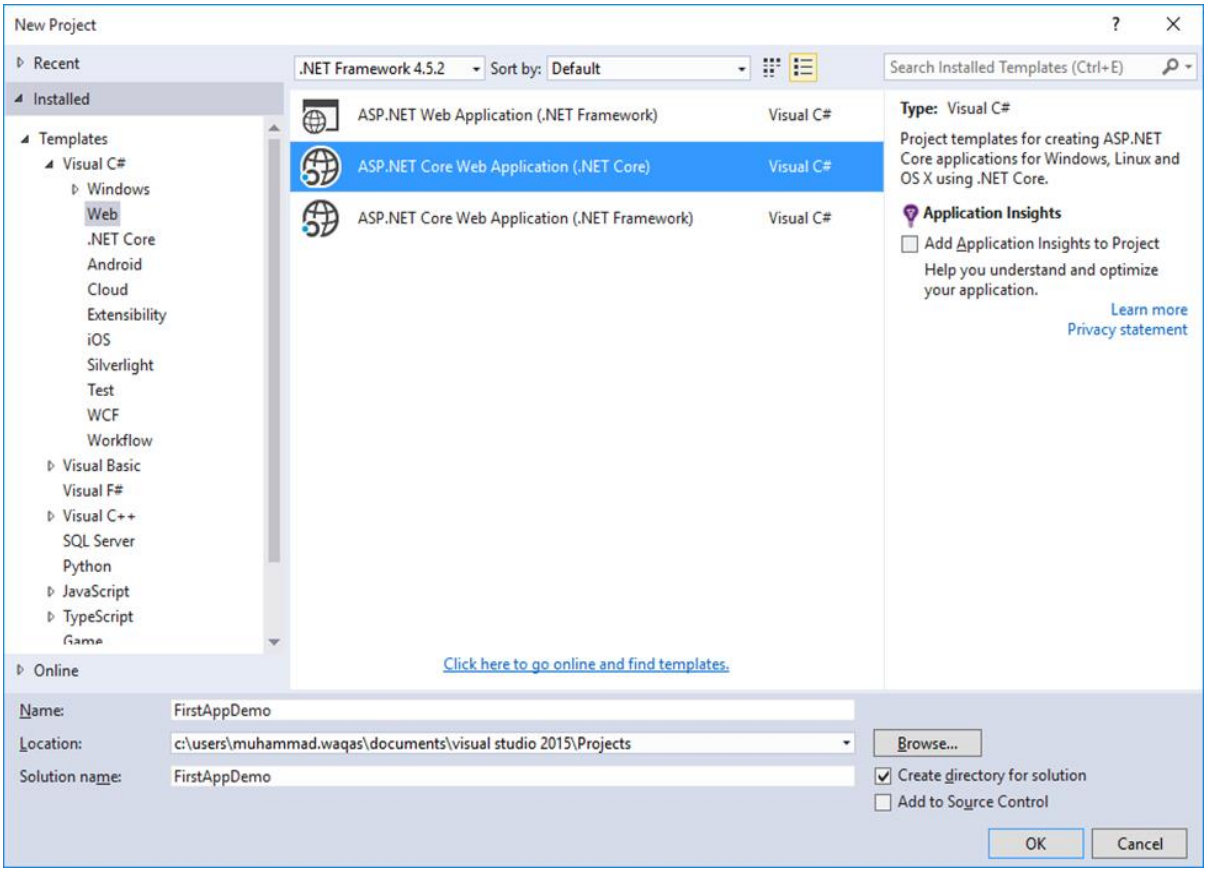


Step 10: You are now ready to start your application using ASP.NET Core.

3. ASP.NET Core — New Project

In this chapter, we will discuss how to create a new project in Visual Studio.

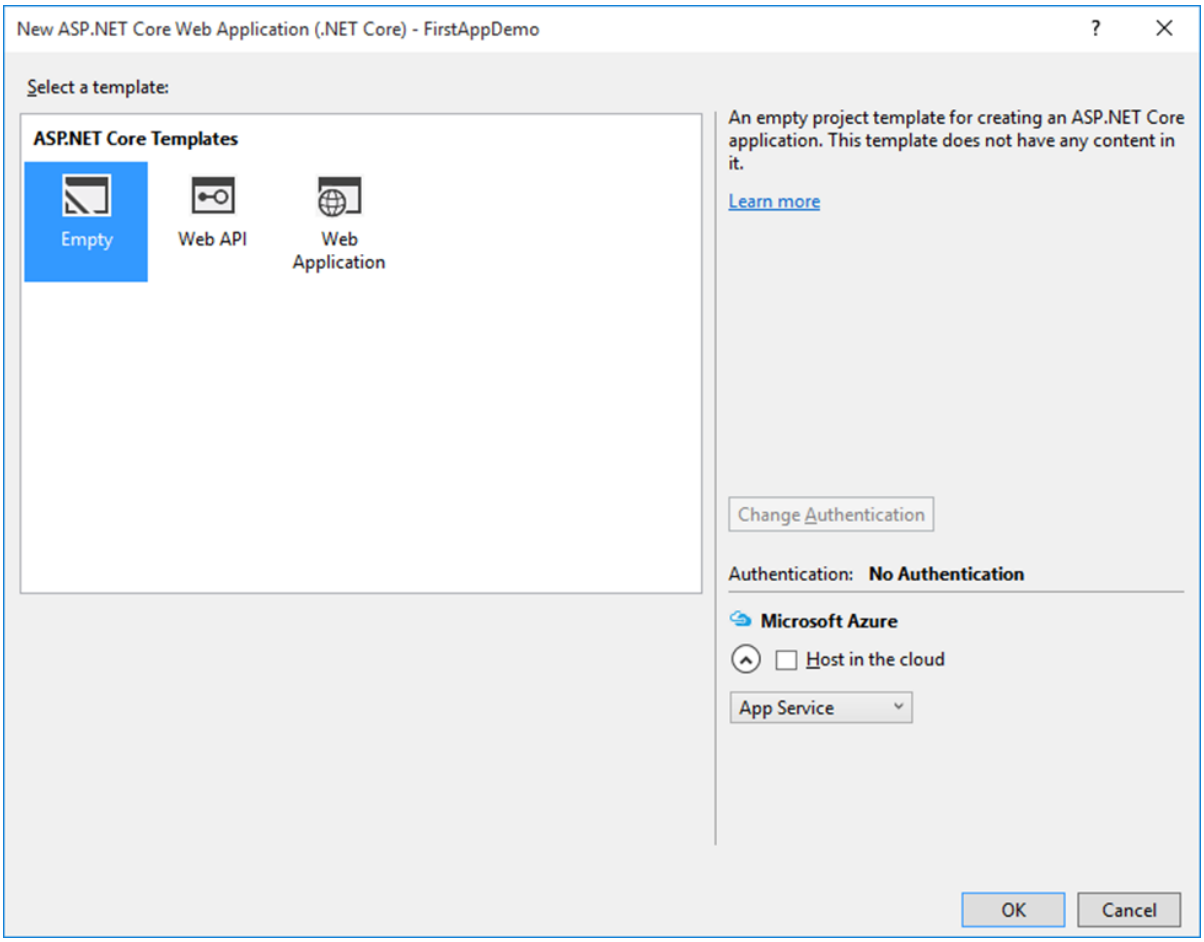
Once you have installed the Visual Studio 2015 tooling, you can start building a new ASP.NET Core Application from the **File > New Project** menu option.



On the New Project dialog box, you will see the following three different templates for Web projects:

- **ASP.NET Web Application:** The simple ASP.NET application templates
- **ASP.NET Core Web Application (.NET Core):** This will start you with a cross-platform compatible project that runs on the .NET Core framework
- **ASP.NET Core Web Application (.NET Framework):** This starts a new project that runs on the standard .NET Framework on Windows.

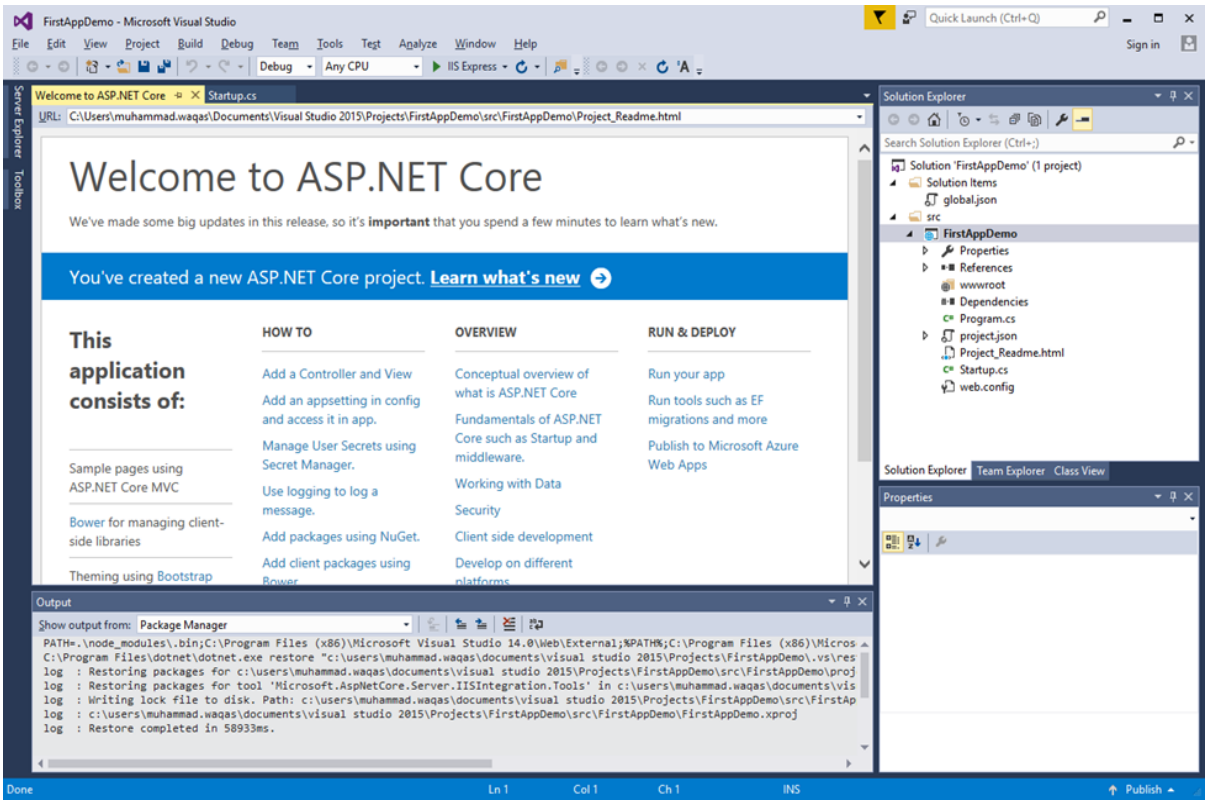
In the left pane, select **Templates > Visual C# > Web** and in the middle pane select the ASP.NET Core Web Application (.NET Core) template. Let us call this application **FirstAppDemo** and also specify the Location for your ASP.NET Core project and then Click OK.



In the above dialog box, you can select a specific template for the ASP.NET application from the available ASP.NET Core Templates.

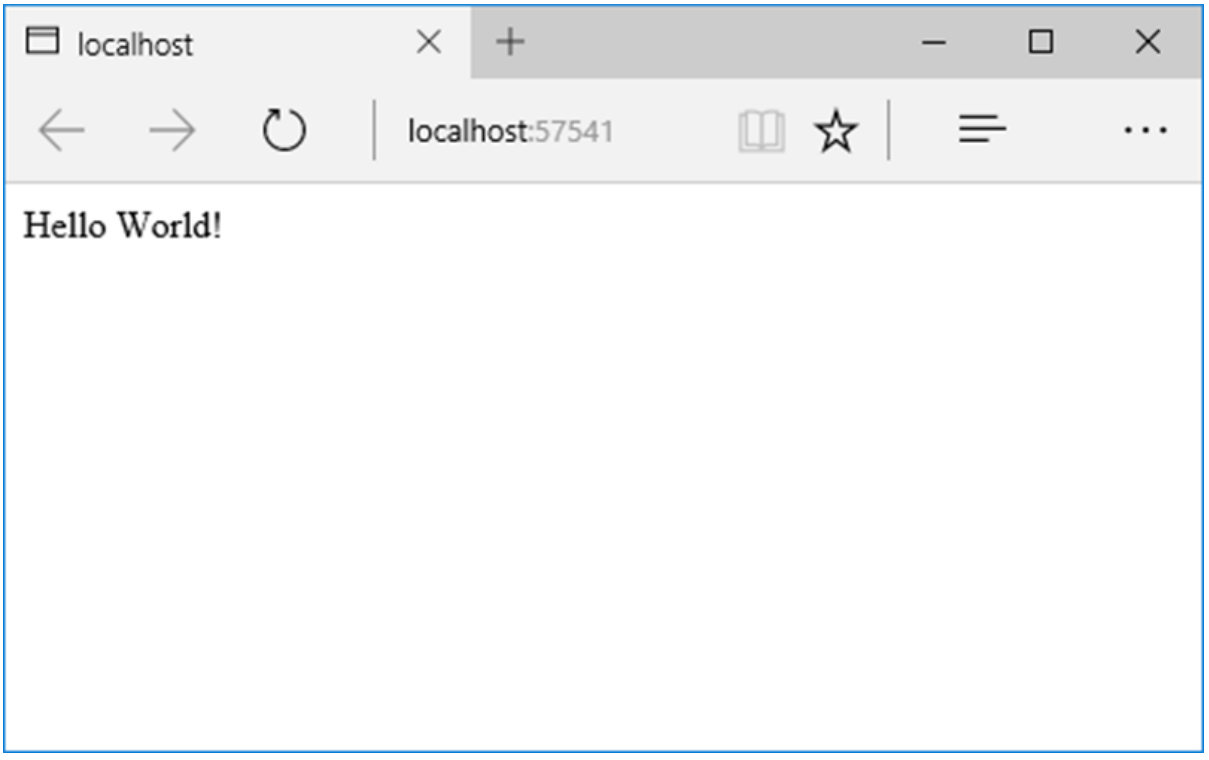
ASP.NET Core templates currently contain three different templates. Of these, the **Web Application template** will help you lay out a lot of files on your file system. This also allows you to use ASP.NET MVC right away.

Here, we will start with an empty template. This would help us build it from scratch. Let us select the Empty template, turn off the Host in the cloud and click OK.

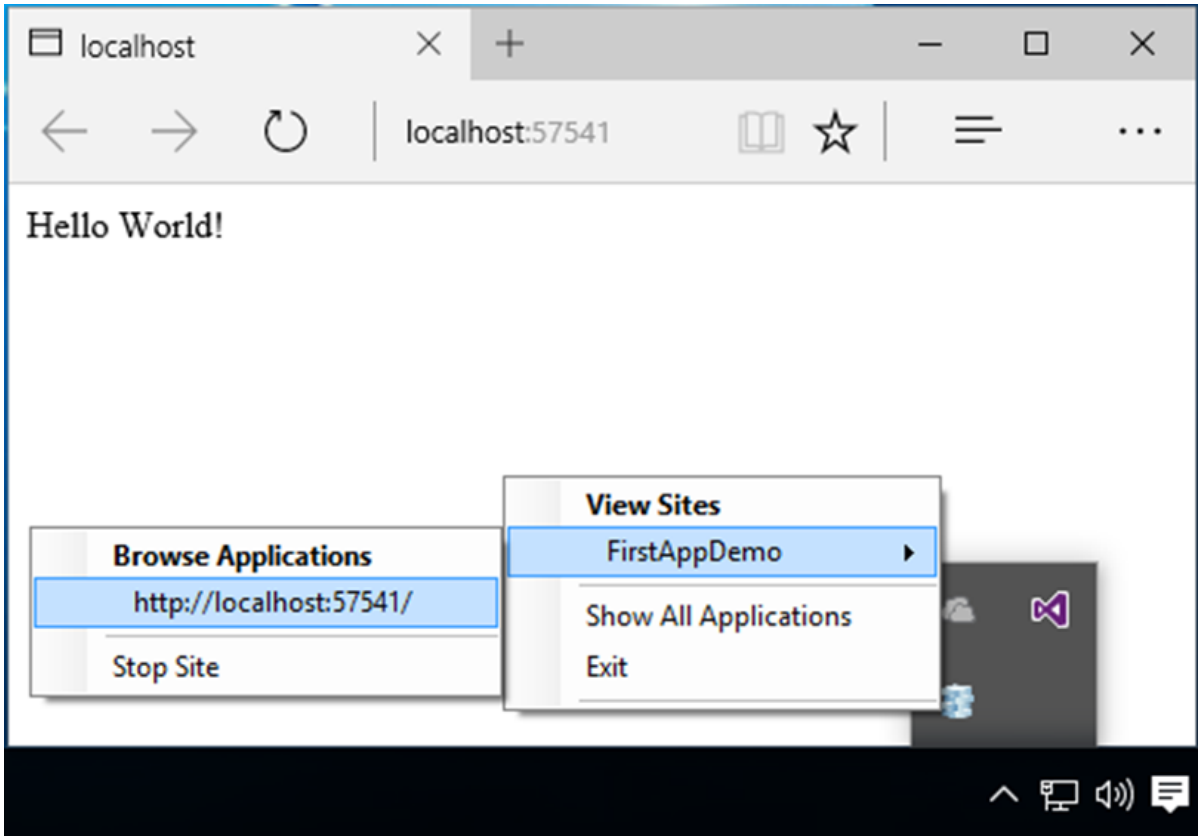


Visual Studio will now launch the project in some time. In the Solution Explorer window, you will see all the files that are in this project.

Let us run this application, you can do that by pressing Ctrl+F5 or by going to the Debug menu. After going to the Debug menu, select **Start Without Debugging**.



This application can display only Hello World! This runs on **localhost:57741**. In the window system tray, you can also see that IIS Express is running.

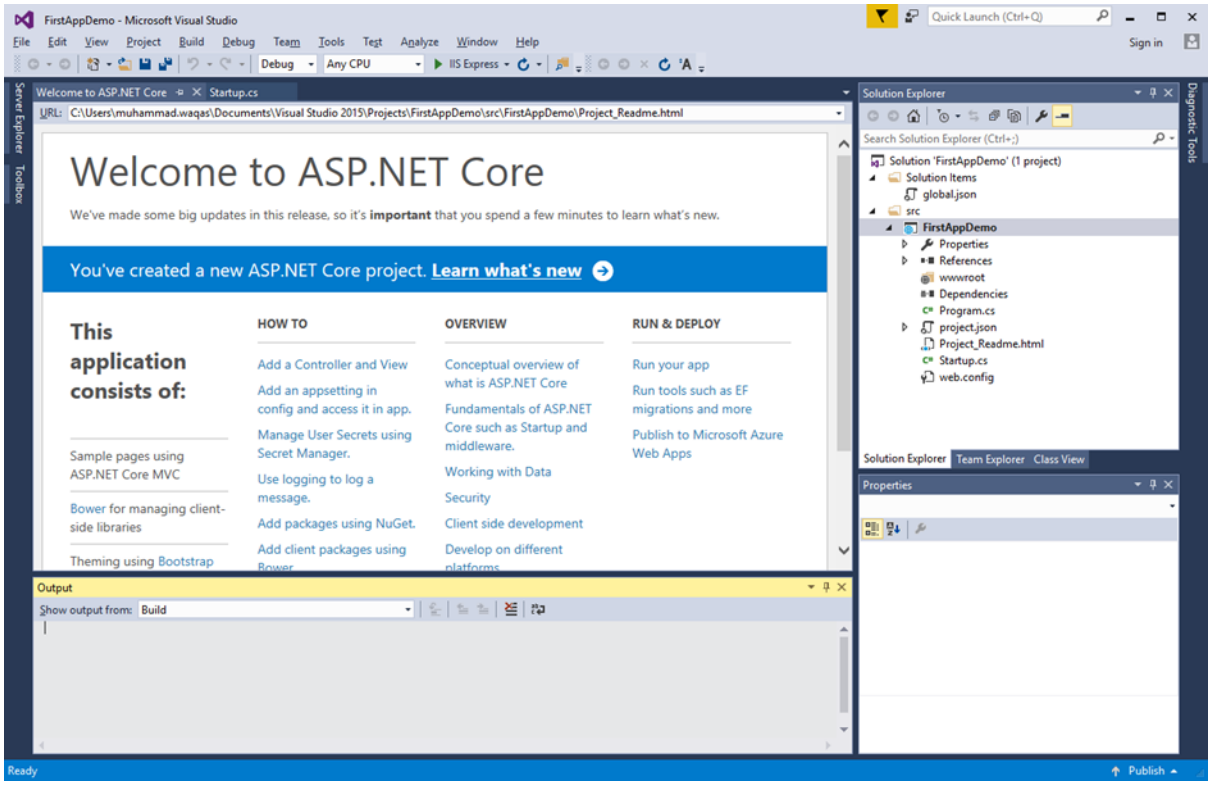


And the name of the site is **FirstAppDemo**. If you have been programming with ASP.NET with the previous versions of the framework, the way you would interact with the Visual Studio and the way Visual Studio uses IIS Express to host your application, all these aspects will be familiar.

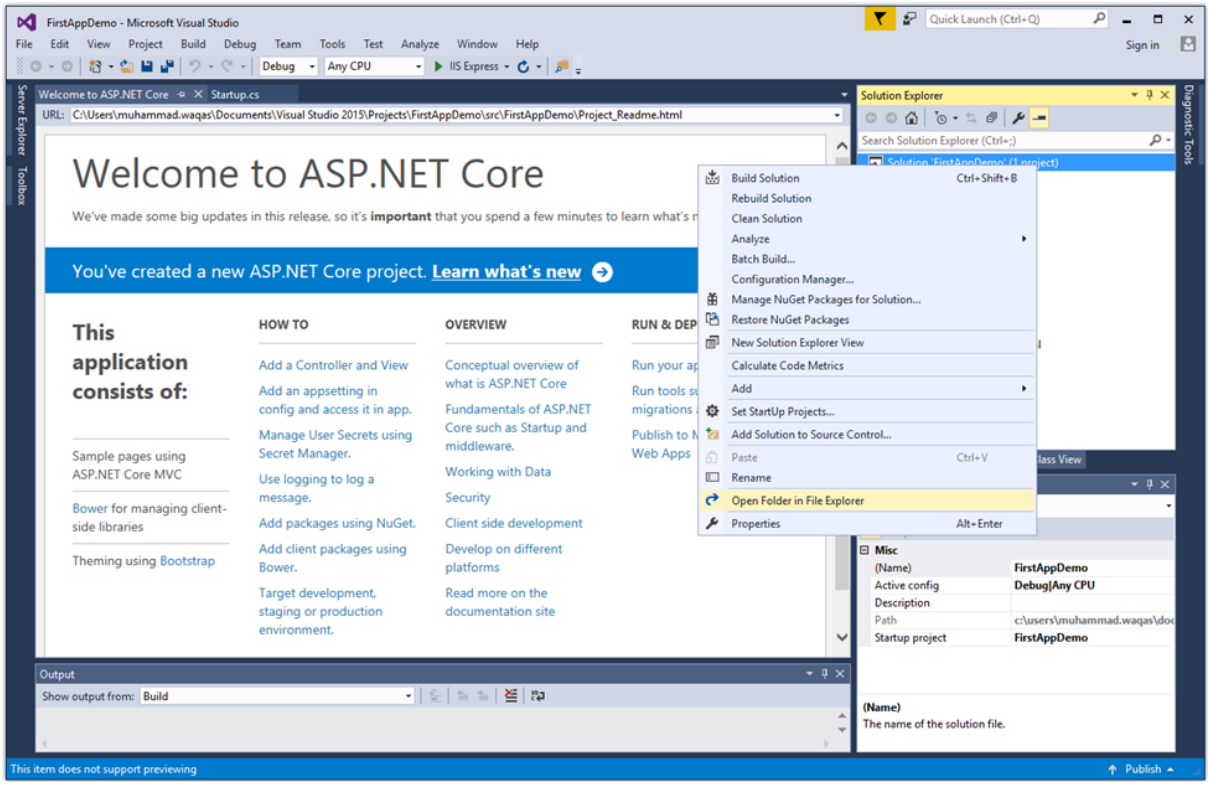
4. ASP.NET Core — Project Layout

In this chapter, we will discuss how ASP.NET core project appears on the file system and how the different files and directories all work together.

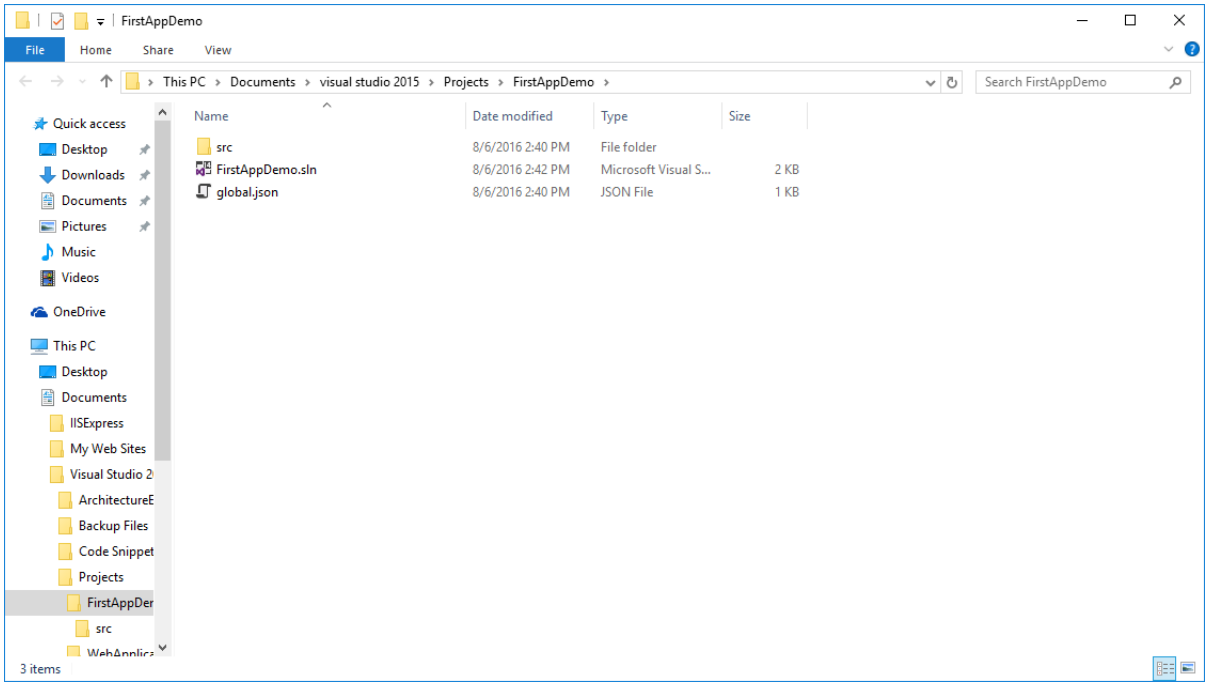
Let us open the **FirstAppDemo** project created in the previous chapter.



In the Solution Explorer window, right-click on the **Solution** node and select the **Open Folder** in **File Explorer**.

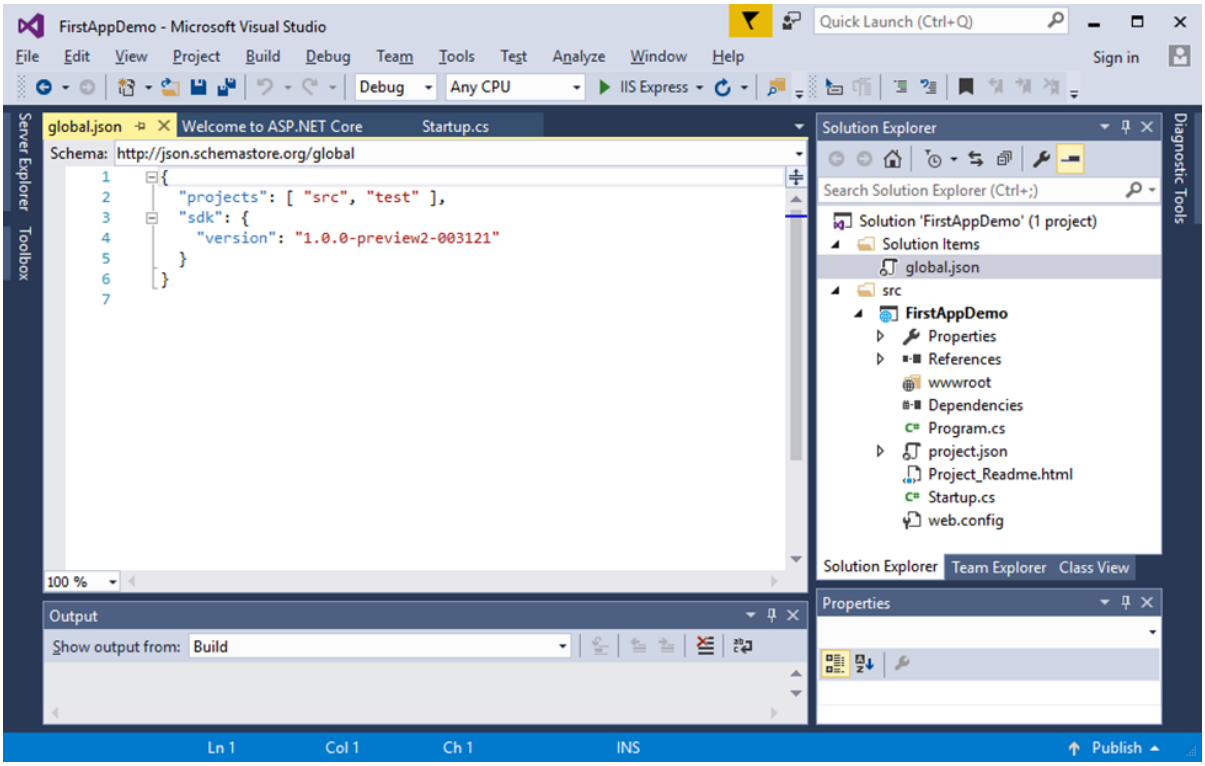


You will see now the root directory with two files in it: **FirstAppDemo.sln** and **global.json**.



FirstAppDemo.sln is a solution file. Visual Studio has used this extension for years by default, and you can double-click on the file if you want to open the app in Studio and work on it.

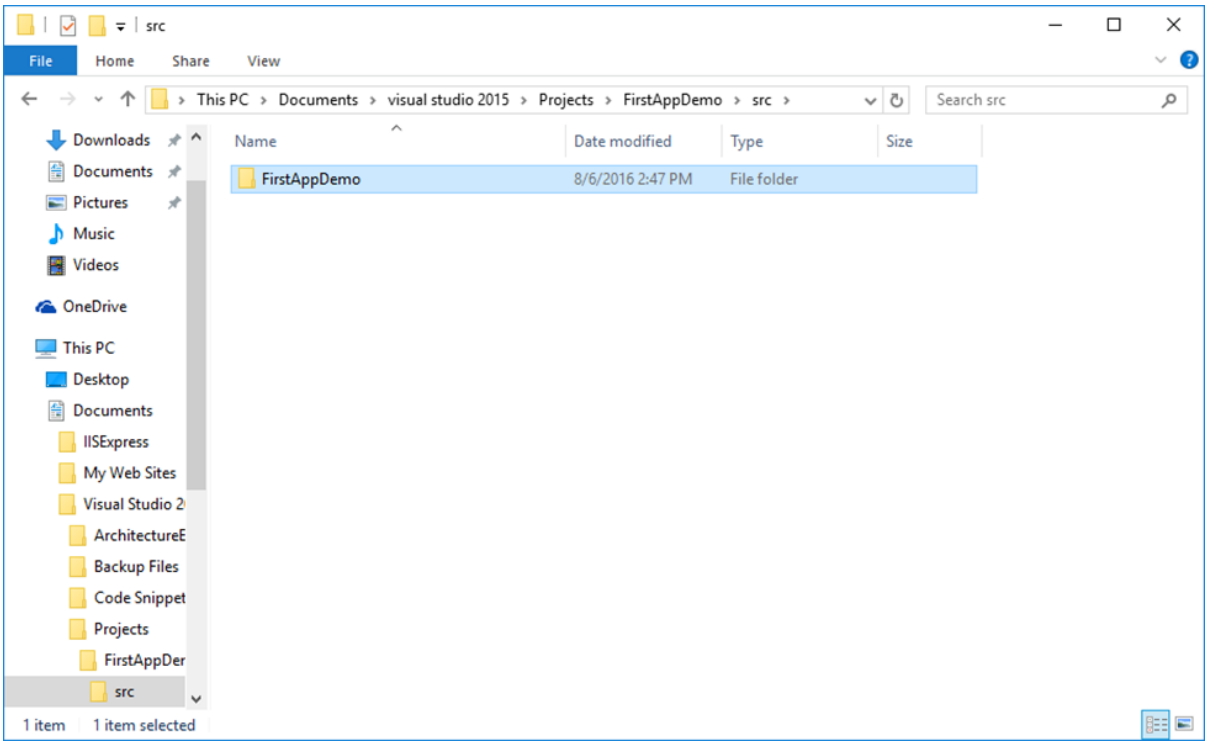
There is also a **global.json** file. Let us open this file in Visual Studio.



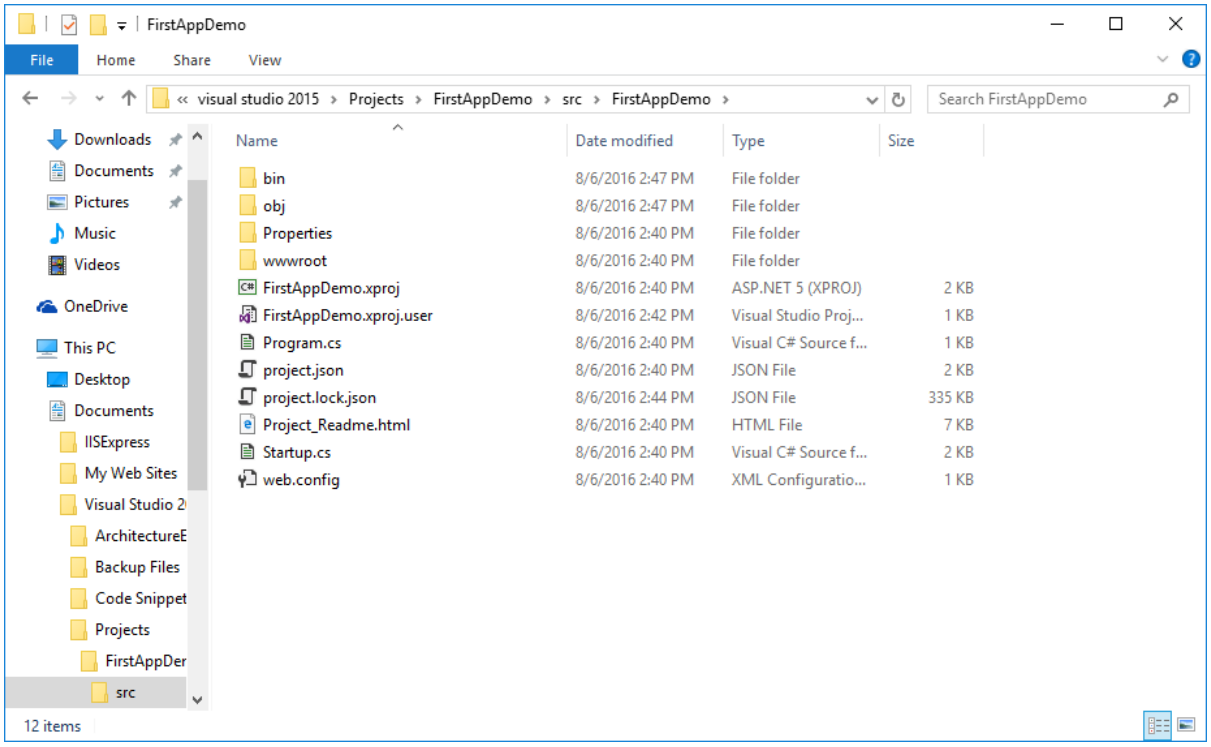
In the file, the project's setting is significant. This project setting tells ASP.NET where to look for your source code and what folders contain your projects.

There are two possible folders **"src"** for source and a **"test"** folder. Unless your projects and source code are in one of these two folders, the code won't be available to build. You can change these settings if you want.

The Windows Explorer has the **"src"** folder on disk. You don't have a test folder. In the test folder, you could place your unit testing projects. Let us double-click on the **"src"** folder.



You can see the FirstAppDemo project, and the web application. Now, double-click on the folder.



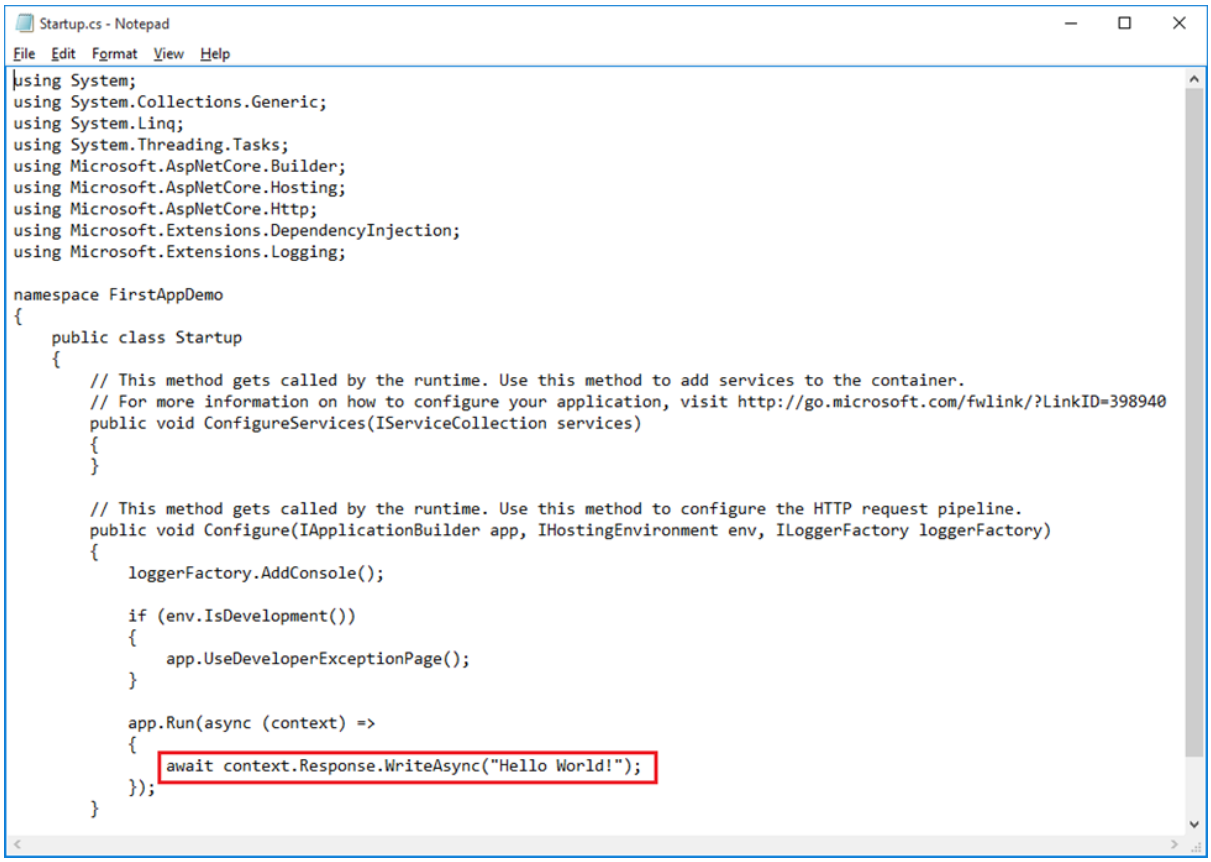
These are the source code files for the application and you can also see this folder structure in the Solution Explorer window. This is because in the present version of ASP.NET Core, the file system determines what is in your project.

If you add a new file to the disk, the file will be added to the project. If you delete a file, the file is removed from the project. Everything stays in sync and that is a little bit different than previous versions of ASP.NET Core where a project file, a *.cs proj file, that contained a manifest of everything that is in the project.

ASP.NET Core also compiles your application when a file changes or a new file appears.

Example

Let us see a simple example by opening the **Startup.cs** file in the text editor.



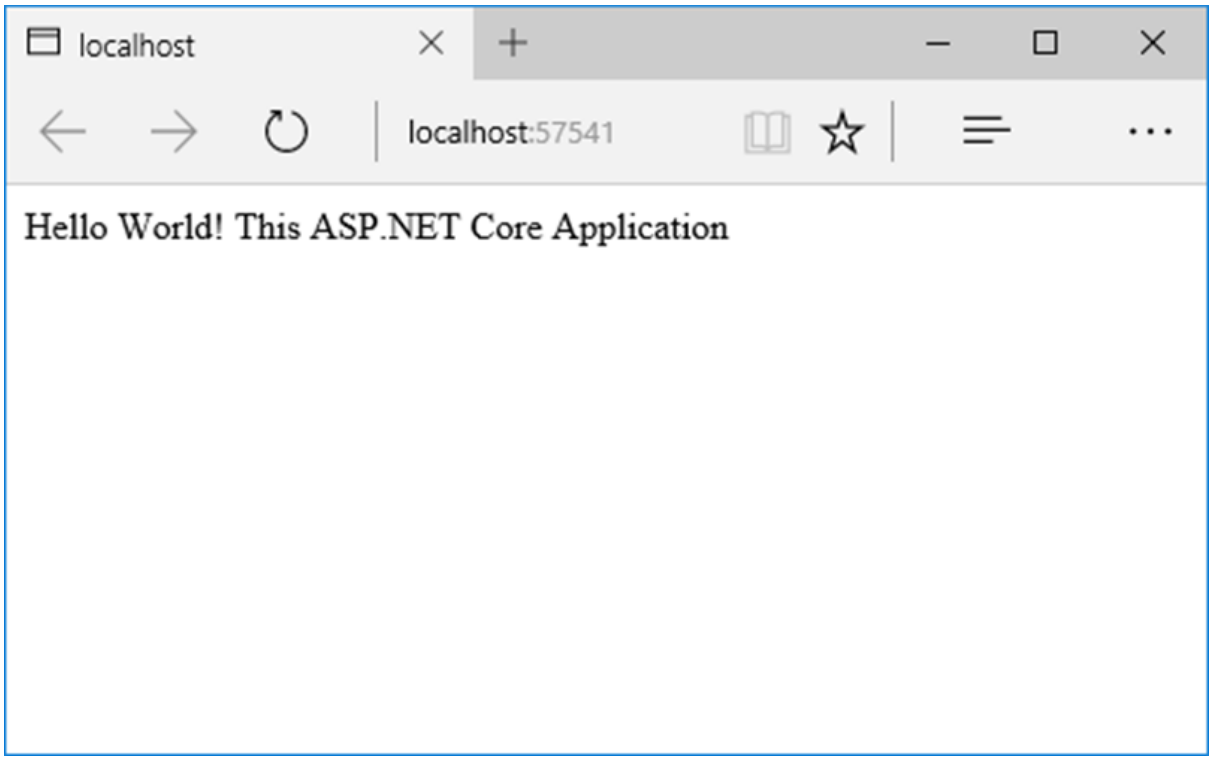
It is this line of code that responds to every HTTP request to your application and it simply responds with Hello World!

Let us change the string in the above screenshot by saying **"Hello World! This ASP.NET Core Application"** as shown in the following program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.Extensions.Logging;


namespace FirstAppDemo
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }


        // This method gets called by the runtime. Use this method to configure
the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();


            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }


            app.Run(async (context) =>
            {
                await context.Response.WriteAsync("Hello World! This ASP.NET
Core Application");
            });
        }
    }
}
```

Save this file in the text editor by pressing Ctrl + S and then go back to the web browser and refresh the application.



You can now see that your changes are reflected in the browser.

- This is because ASP.NET will monitor the file system and automatically recompile the application when a file changes. You don't need to explicitly build the application in Visual Studio.

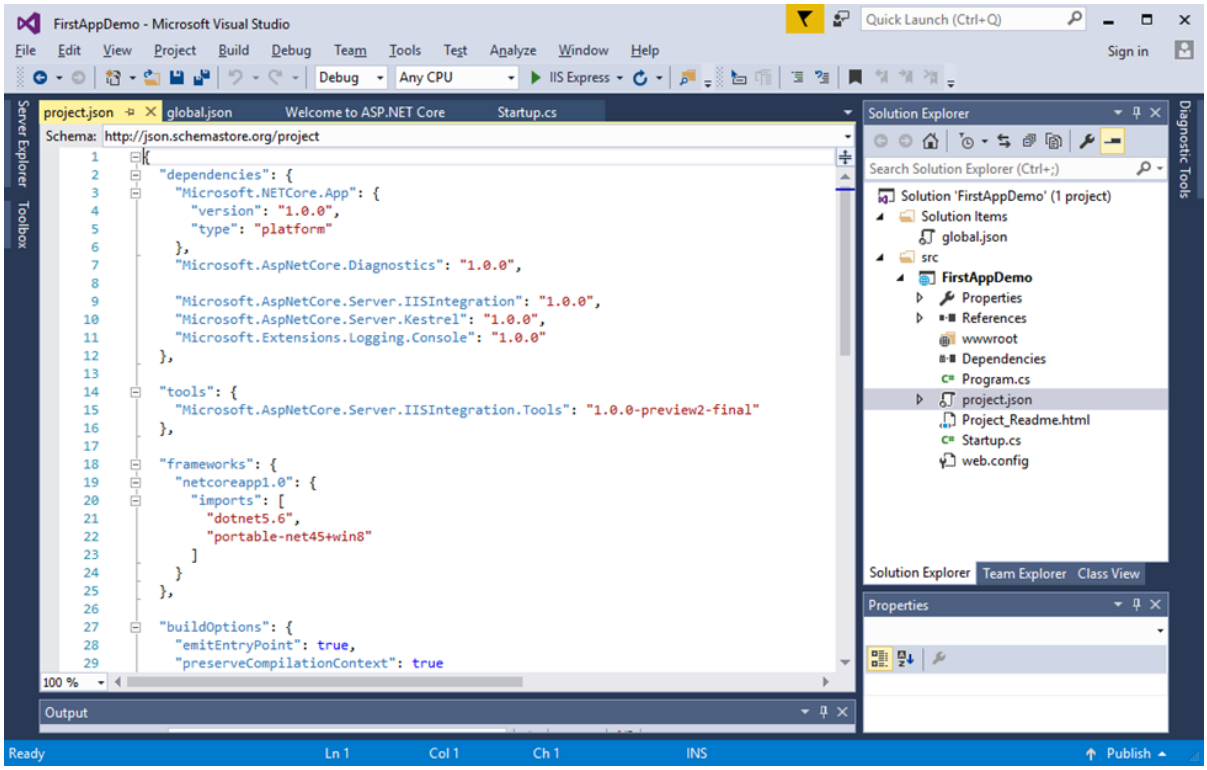
- In fact, you can use a completely different editor, something like Visual Studio Code.

- All you need to do with the Visual Studio is get the web server started by running without the debugger. You can also press Ctrl + F5, and can edit files, save files, and just refresh the browser to see the changes.

- This is a nice workflow for building web applications with a compiled language like C#.

5. ASP.NET Core — Project.json

In this chapter, we will discuss the **project.json** file. This file is using JavaScript object notation to store configuration information and it is this file that is really the heart of a .NET application. Without this file, you would not have an ASP.NET Core project. Here, we will discuss some of the most important features of this file. Let us double-click on the **project.json** file.



Currently, the default code implementation in project.json file is as follows:

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",

    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0"
  },
```

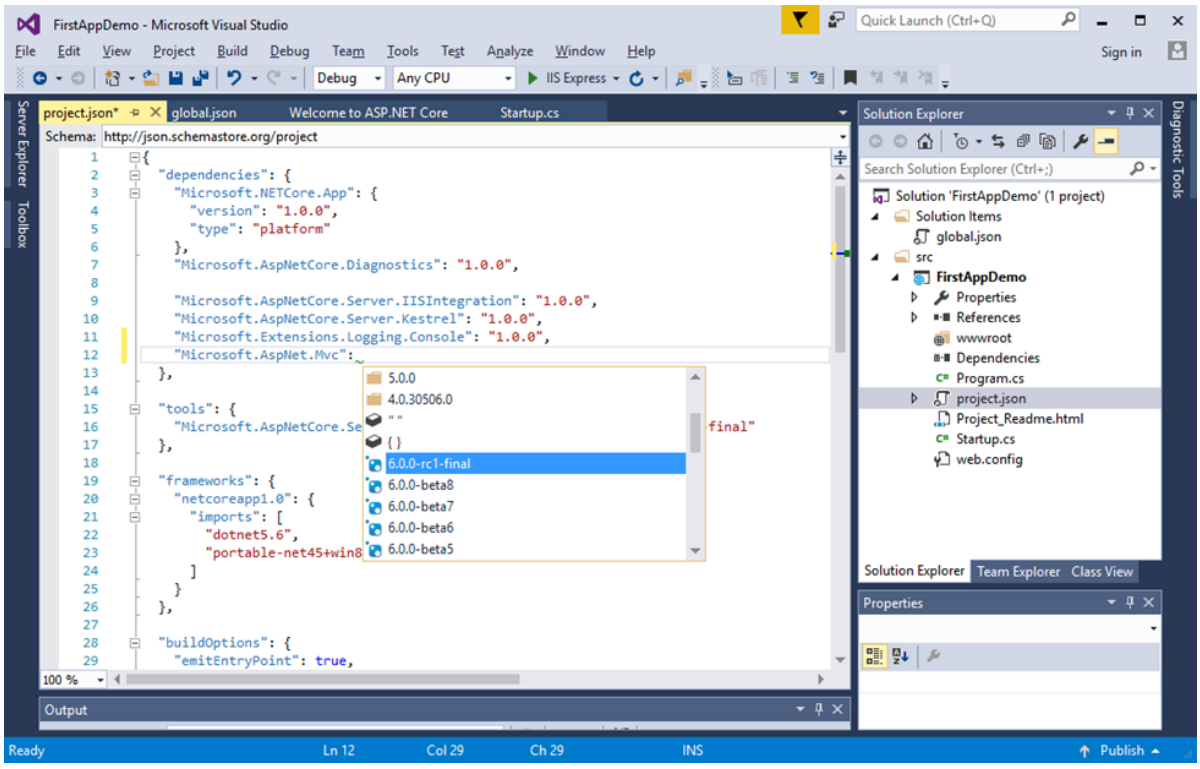
```
  "tools": {
    "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
  },

  "frameworks": {
    "netcoreapp1.0": {
      "imports": [
        "dotnet5.6",
        "portable-net45+win8"
      ]
    }
  },

  "buildOptions": {
    "emitEntryPoint": true,
    "preserveCompilationContext": true
  },

  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true
    }
  },

  "publishOptions": {
    "include": [
      "wwwroot",
      "web.config"
    ]
  },

  "scripts": {
    "postpublish": [ "dotnet publish-iis --publish-folder %publish:OutputPath%
--framework %publish:FullTargetFramework%" ]
  }
}
```

As we see, we have version information at the top of this file. This is the version number your application will use when you build it.

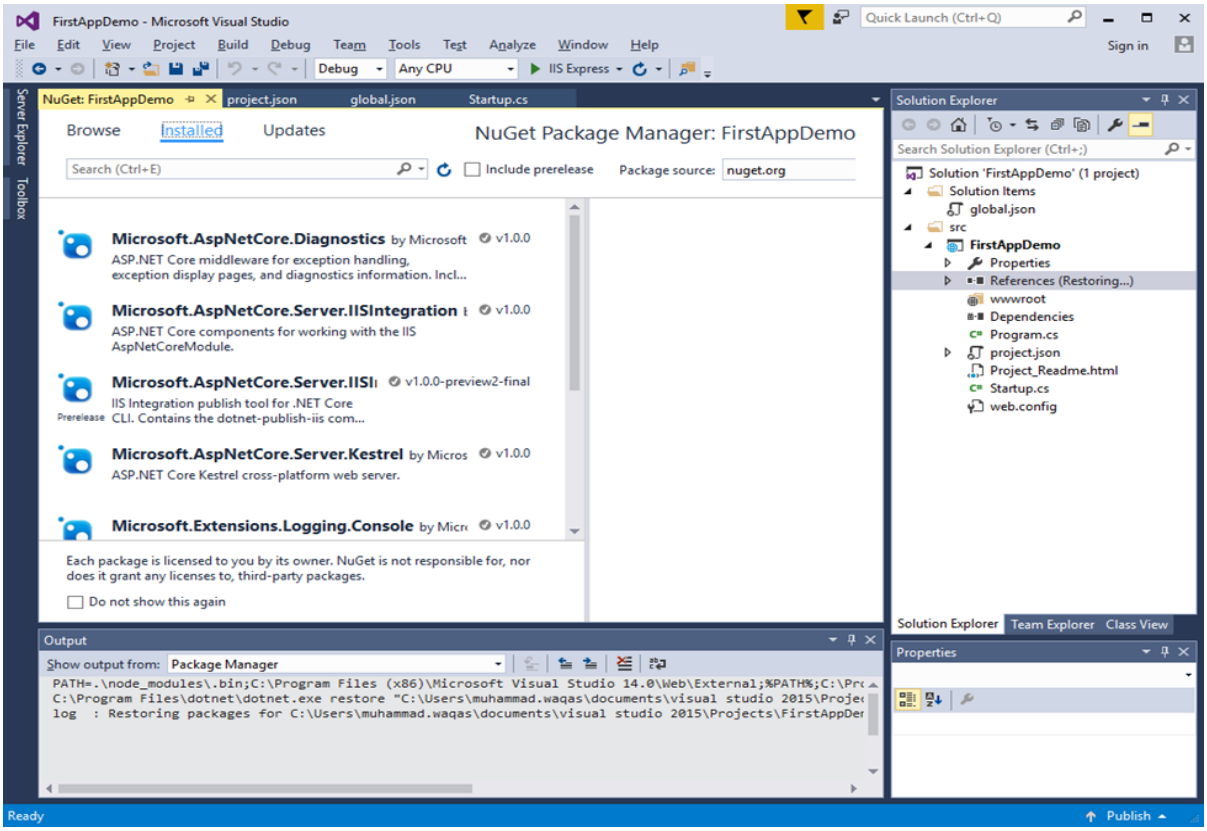
- The version is 1.0.0, but the most important part of this file is the dependencies.
- If your application is going to do any useful work, then you will need libraries and frameworks to do that work, such storing and retrieving data to/from a database or render complicated HTML.
- With this version of ASP.NET Core, the dependencies are all managed via the NuGet package manager.
- NuGet has been around the .NET space for a few years, but now the primary way to manage all your dependencies is by using libraries and frameworks that are wrapped as NuGet packages.
- All of the top-level NuGet packages your application needs will be stored in this project.json file.

```
"Microsoft.AspNetCore.Diagnostics": "1.0.0",
"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
"Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
"Microsoft.Extensions.Logging.Console": "1.0.0"
```

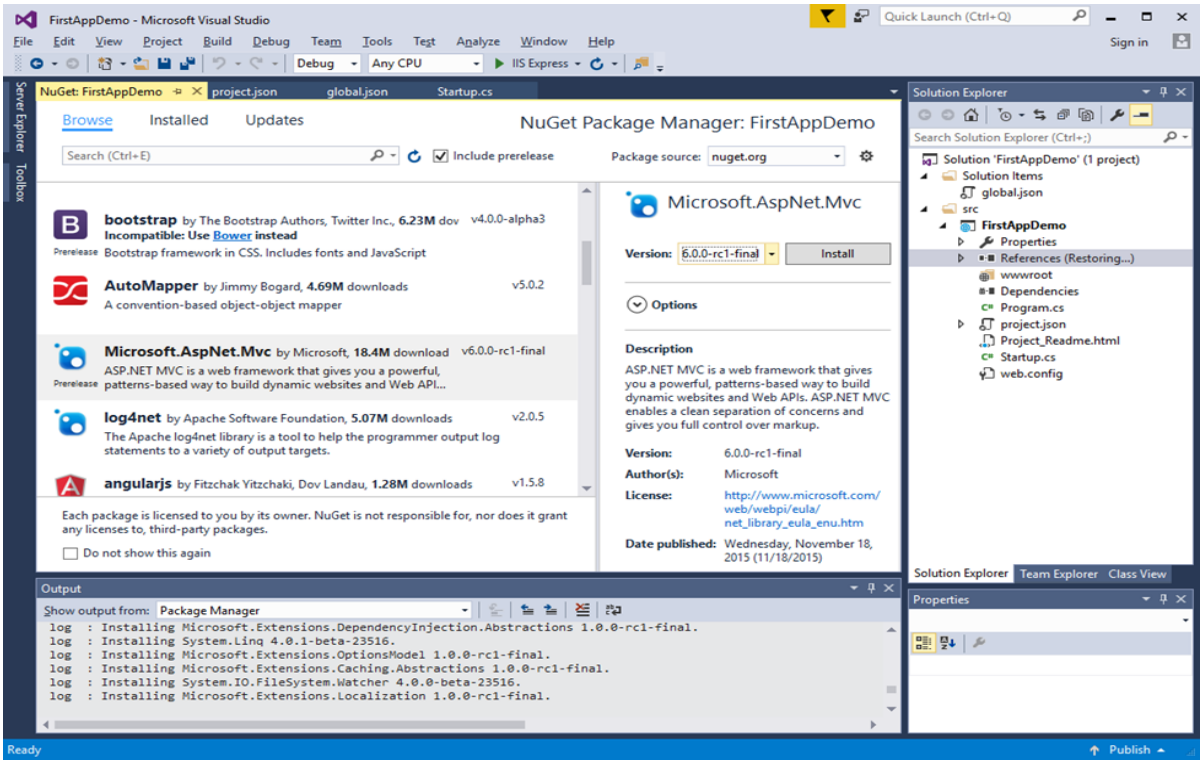
You can see we have some dependencies in this file and the exact dependencies will probably change by the final release of ASP.NET. When you want to add a new dependency, say like the ASP.NET MVC framework, you easily type into this project.json file, and you will also get some **IntelliSense** help including not just the package name but also the version numbers as shown in the following screenshot.



You can also use the UI by right-clicking on References in the Solution Explorer and then, select Manage NuGet packages. You can now see the currently installed packages.



Those packages are the same ones that are in your project.json file and you can also go to the Browser and add other packages, including pre-released packages, let us say, the MVC framework installed into this project.



If you install this package right now by using the Install button, then this package would be stored in project.json. The frameworks section is another important part of project.json, this section tells ASP.NET as to which of the .NET frameworks your application can use.

```
"frameworks": {
    "netcoreapp1.0": {
      "imports": [
        "dotnet5.6",
        "portable-net45+win8"
      ]
    }
  },
```

In this case, you will see that **"netcoreapp1.0"** is the framework used in the project, you can also include the full .NET Framework which is installed when you install Visual Studio.

- It is the same .NET Framework that is included with many versions of the Windows Operating System.
- It is the .NET Framework that has been around for 15 years, and it includes the frameworks that do everything from web programming to desktop programming.
- It is a huge framework that works only on Windows.
- "**netcoreapp1.0**" is the .NET Core framework. It is a cross-platform framework and can work on various platforms, not just Windows but also OS X and Linux.
- This framework has fewer features than the full .NET framework, but it does have all the features that we need for ASP.NET Core web development.

6. ASP.NET Core — Configuration

In this chapter, we will be discussing the configuration related to ASP.NET Core project. In Solution Explorer, you will see the Startup.cs file. If you have worked with previous versions of ASP.NET Core, you will probably expect to see a global.asax file, which was one place where you could write codes to execute during startup of a web application.

- You would also expect to see a web.config file containing all the configuration parameters your application needed to execute.

- In ASP.NET Core those files are all gone, and instead of configuration and startup code are loaded from Startup.cs.

- There is a Startup class inside the file and in this class you can configure your application and even configure your configuration sources.

Here is the default implementation in the **Startup.cs** file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace FirstAppDemo
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime. Use this method to configure
the HTTP request pipeline.
```

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.Run(async (context) =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        }
    }
}
```

In the Startup class, there are two methods where most of our work will take place. The Configure method of the class is where you build your HTTP processing pipeline.

- This defines how your application responds to requests. Currently this application can only say Hello World! and if we want the application to behave differently, we will need to change the pipeline around by adding additional code in this Configure method.

- For example, if we want to serve the static files such as an index.html file, we will need to add some code to the Configure method.

- You can also have an error page or route requests to an ASP.NET MVC controller; both of these scenarios will also require to do some work in this Configure method.

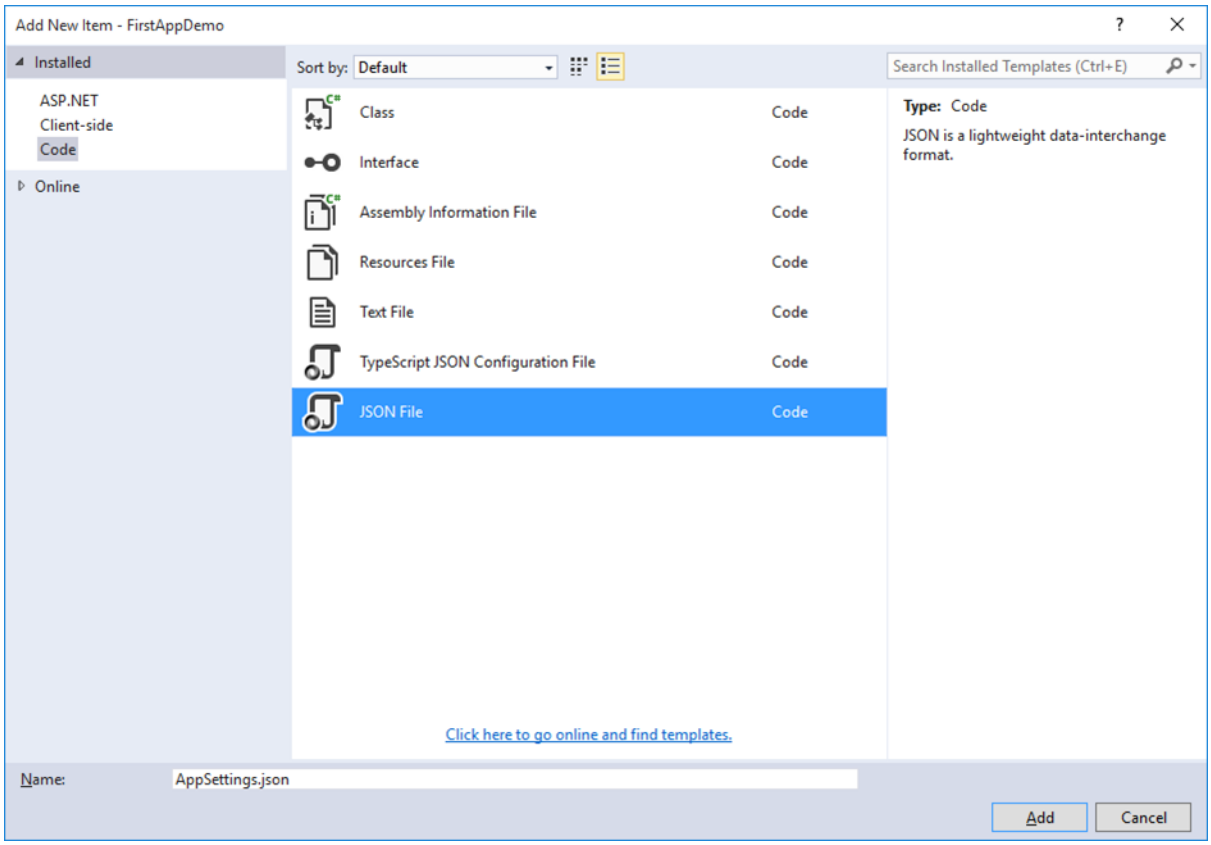
- In the Startup class, you will also see the **ConfigureServices()** method. This helps you configure components for your application.

Right now, we have a hard-coded string for every response — the **Hello World!** string. Instead of hard-coding the string, we want to load this string from some component that knows the text that we want to display.

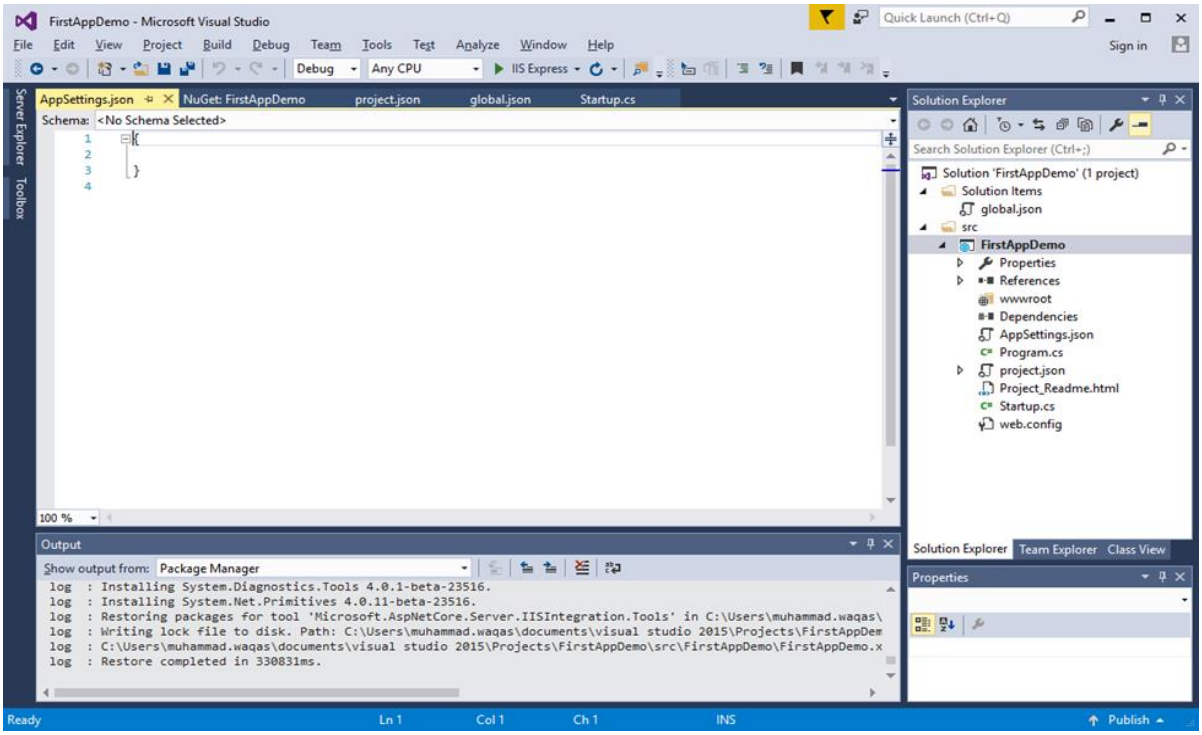
- This other component might load that text from a database or a web service or a JSON file, it doesn't matter where exactly it is.

- We will just set up a scenario so that we do not have this hard-coded string.

In the Solution Explorer, right-click on your project node and select **Add > New Item**.



In the left pane, select **Installed > Code** and then in the middle pane, select the JSON File. Call this file **AppSettings.json** and click on the **Add** button as in the above screenshot.



We can also have our program read the text from the file instead of having the Hello World! String in Startup.cs. Let us add the following code in **AppSettings.json** file.

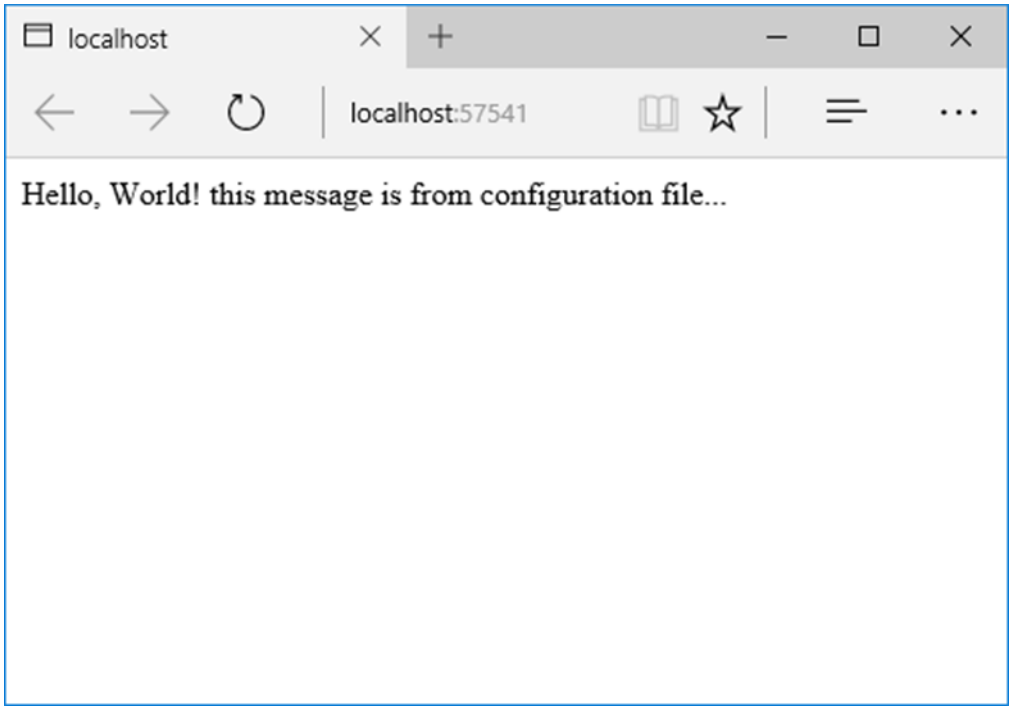
```
{
  "message": "Hello, World! this message is from configuration file..."
}
```

Now we need to access this message from the Startup.cs file. Here is the implementation of the **Startup.cs** file which will read the above message from the JSON file.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace FirstAppDemo
{
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
      public void Configure(IApplicationBuilder app)
        {
```

```
            app.UseIISPlatformHandler();

            app.Run(async (context) =>
            {
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });

        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Let us now run the application. Once you run the application, it will produce the following output.



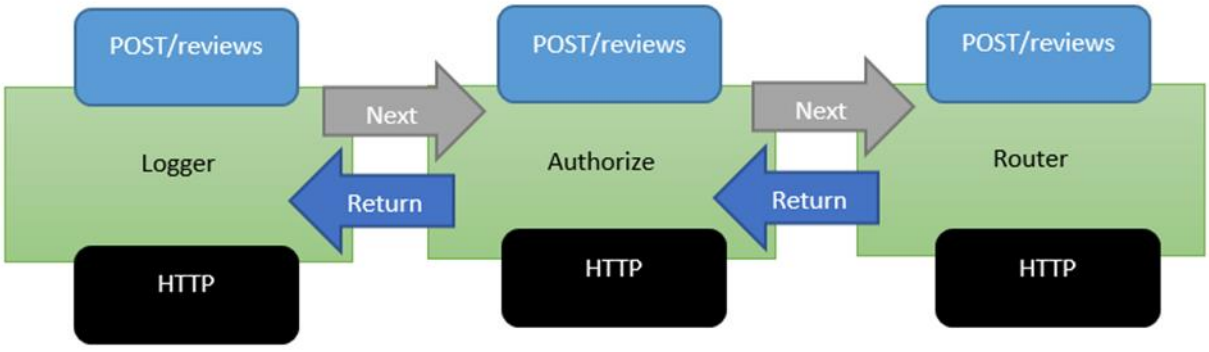
7. ASP.NET Core — Middleware

In this chapter, we will understand how to set up middleware. Middleware in ASP.NET Core controls how our application responds to HTTP requests. It can also control how our application looks when there is an error, and it is a key piece in how we authenticate and authorize a user to perform specific actions.

- Middleware are software components that are assembled into an application pipeline to handle requests and responses.
- Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline.
- Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.
- Each piece of middleware in ASP.NET Core is an object, and each piece has a very specific, focused, and limited role.
- Ultimately, we need many pieces of middleware for an application to behave appropriately.

Let us now assume that we want to log information about every request into our application.

- In that case, the first piece of middleware that we might install into the application is a logging component.
- This logger can see everything about the incoming request, but chances are a logger is simply going to record some information and then pass along this request to the next piece of middleware.



- Middleware is a series of components present in this processing pipeline.
- The next piece of middleware that we've installed into the application is an authorizer.

- An authorizer might be looking for specific cookie or access tokens in the HTTP headers.
- If the authorizer finds a token, it allows the request to proceed. If not, perhaps the authorizer itself will respond to the request with an HTTP error code or redirect code to send the user to a login page.
- But, otherwise, the authorizer will pass the request to the next piece of middleware which is a router.
- A router looks at the URL and determines your next step of action.
- The router looks over the application for something to respond to and if the router doesn't find anything to respond to, the router itself might return a **404 Not Found error**.

Example

Let us now take a simple example to understand more about middleware. We set up the middleware in ASP.NET using the Configure method of our **Startup class**.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace FirstAppDemo
{
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
```

```
        {
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.Run(async (context) =>
            {
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });

        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Inside the **Configure()** method, we will invoke the extension methods on the IApplicationBuilder interface to add middleware.

There are two pieces of middleware in a new empty project by default:

- IISPlatformHandler
- Middleware registered with app.Run

IISPlatformHandler

IISPlatformHandler allows us to work with Windows authentication. It will look at every incoming request and see if there is any Windows identity information associated with that request and then it calls the next piece of middleware.

Middleware registered with app.Run

The next piece of middleware in this case is a piece of middleware registered with **app.Run**. The Run method allows us to pass in another method, which we can use to process every single response. Run is not something that you will see very often, it is something that we call a terminal piece of middleware.

Middleware that you register with Run will never have the opportunity to call another piece of middleware, all it does is receive a request, and then it has to produce some sort of response.

You also get access to a Response object and one of the things you can do with a Response object is to write a string.

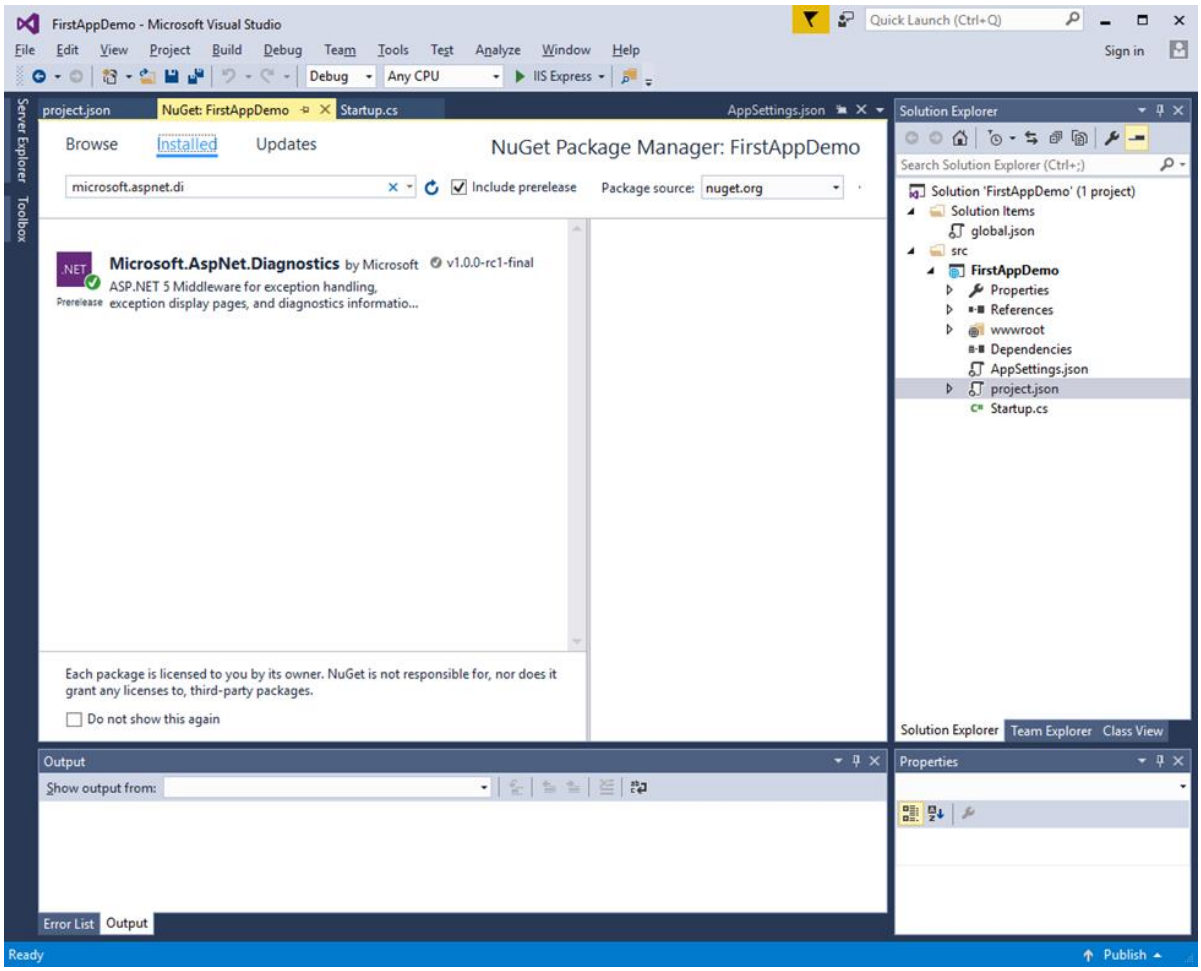
If you want to register another piece of middleware after app.Run, that piece of middleware would never be called because, again, Run is a terminal piece of middleware. It will never call into the next piece of middleware.

How to Add another Middleware

Let us proceed with the following steps to add another middleware:

Step 1: To add another middleware, right-click on project and select Manage NuGet Packages.

Step 2: Search for **Microsoft.aspnet.diagnostics** that is actually ASP.NET Core middleware for exception handling, exception display pages, and diagnostics information. This particular package contains many different pieces of middleware that we can use.

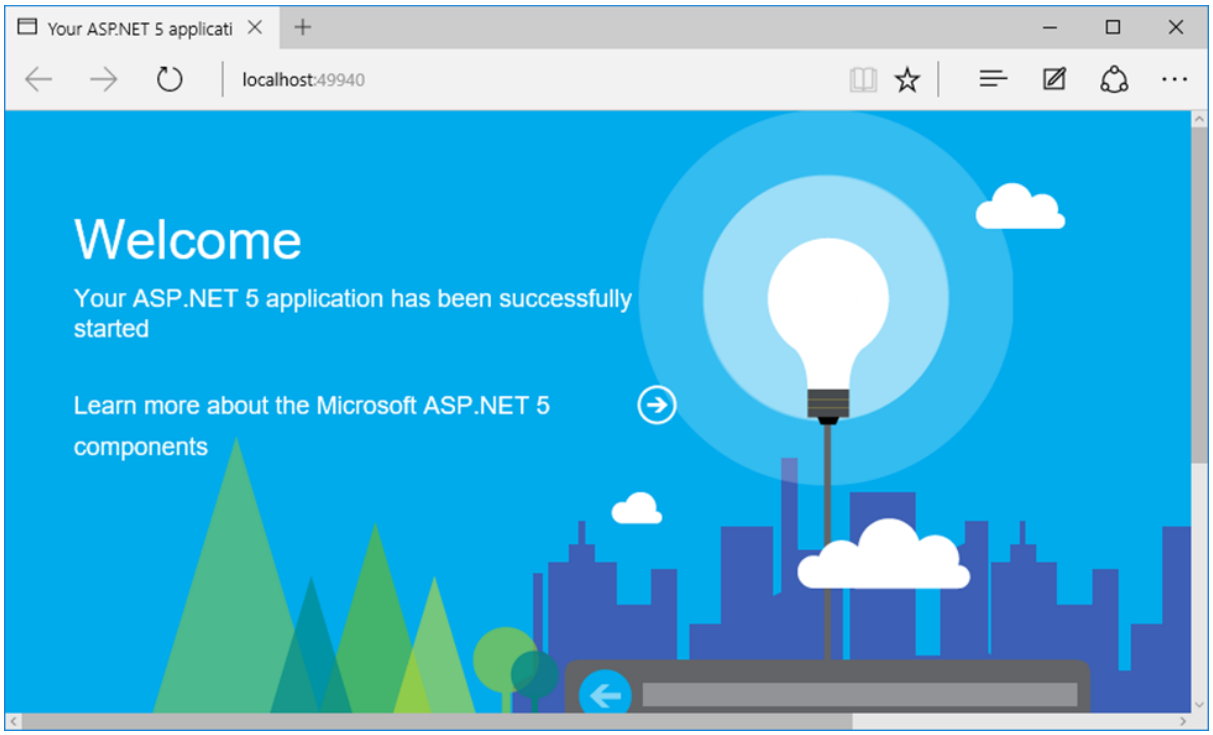


Step 3: Install that package if it is not installed in your project.

Step 4: Let us now go to the **Configure()** method and invoke **app.UseWelcomePage** middleware.

```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();
    app.UseWelcomePage();

    app.Run(async (context) =>
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

Step 5: Run your application and you will see the following welcome screen.

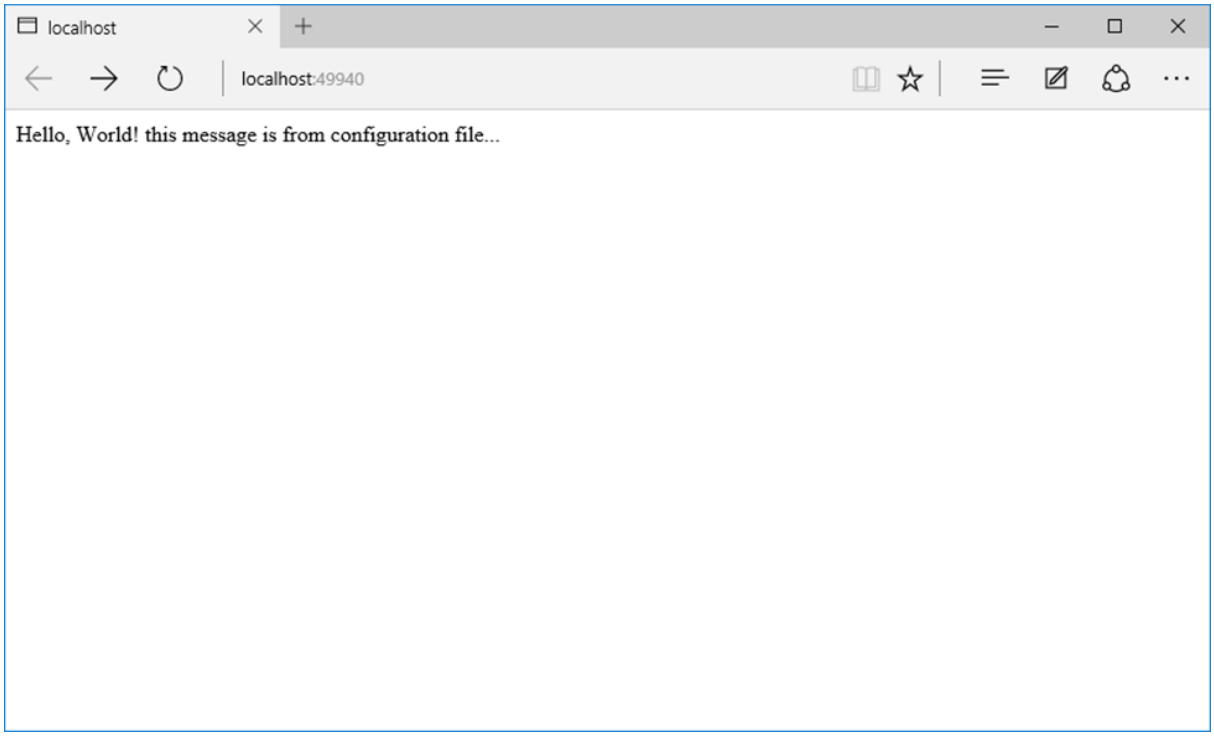


This Welcome screen might not be as useful.

Step 6: Let us try something else that might be a little more useful. Instead of using the Welcome page, we will use the **RuntimeInfoPage**.

```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();
    app.UseRuntimeInfoPage();

    app.Run(async (context) =>
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

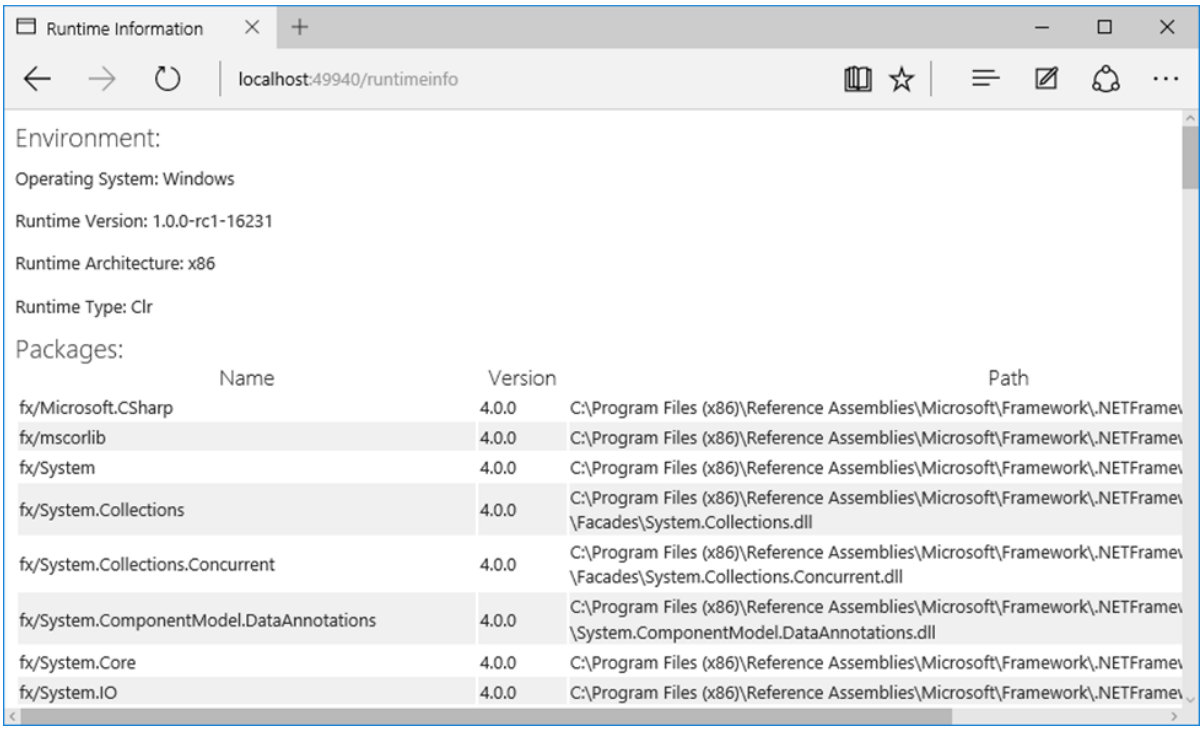
Step 7: Save your **Startup.cs** page and refresh your browser and you will see the following page.



This **RuntimeInfoPage** is a middleware that will only respond to requests that come in for a specific URL. If the incoming request does not match that URL, this piece of middleware just lets the request pass through to the next piece of middleware. The request

will pass through the IISPlatformHandler middleware, then go to the UseRuntimeInfoPage middleware. It is not going to create a response, So it will go to our app.Run and display the string.

Step 8: Let us add **"/runtimeinfo**" at the end of your URL. You will now see a page that is produced by that runtime info page middleware.



You will now see a response that gives you some information about your runtime environment such as the Operating System, runtime version, architecture, type and all the packages that you are using etc.

8. ASP.NET Core — Exceptions

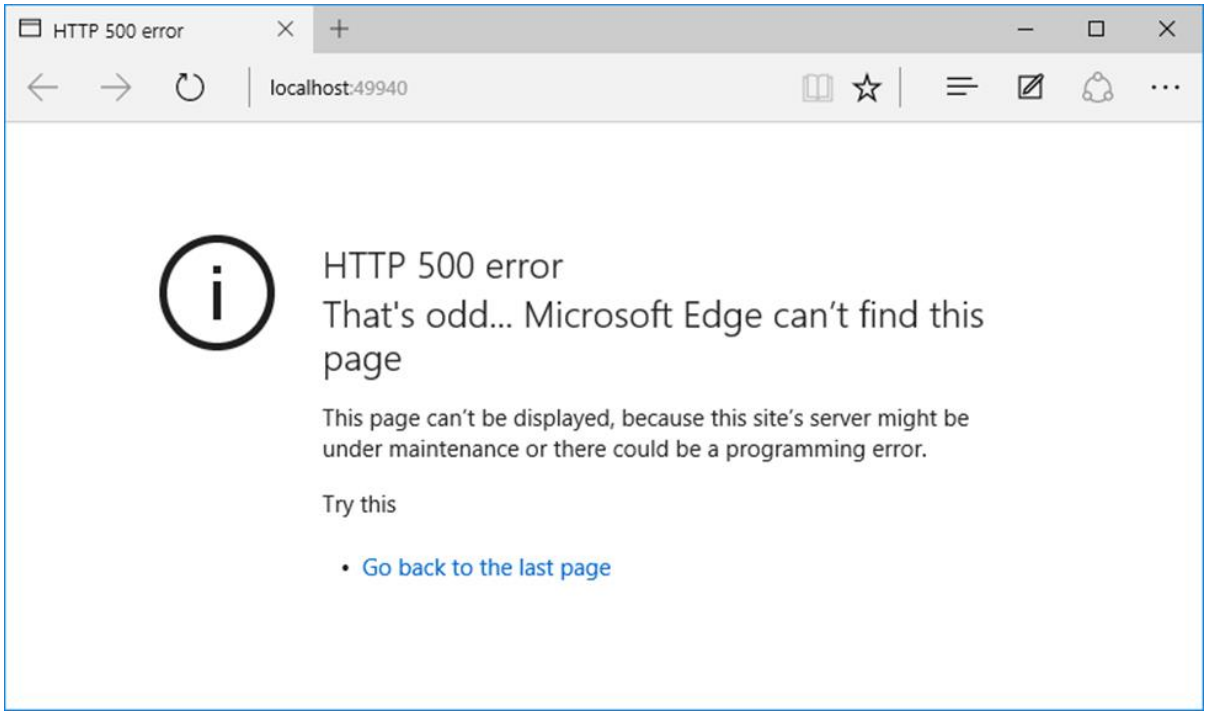
In this chapter, we will discuss **exceptions and error handling**. When errors occur in your ASP.NET Core app, you can handle them in a variety of ways. Let us see an additional piece of middleware that is available through the diagnostics package. This piece of middleware will help us process errors.

To simulate an error, let us go to **app.Run** and see how the application behaves if we just throw an exception every time we hit this piece of middleware.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace FirstAppDemo
{
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime.

        // Use this method to configure the HTTP request pipeline.
```

```
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.UseRuntimeInfoPage();

            app.Run(async (context) =>
            {
                throw new System.Exception("Throw Exception");
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });

        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

It will just throw an exception with a very generic message. Save **Startup.cs** page and run your application.



You will see that we failed to load this resource. There was a HTTP 500 error, an internal server error, and that is not very helpful. It might be nice to get some exception information.

Let us add another piece of middleware, which is the **UseDeveloperExceptionPage**.

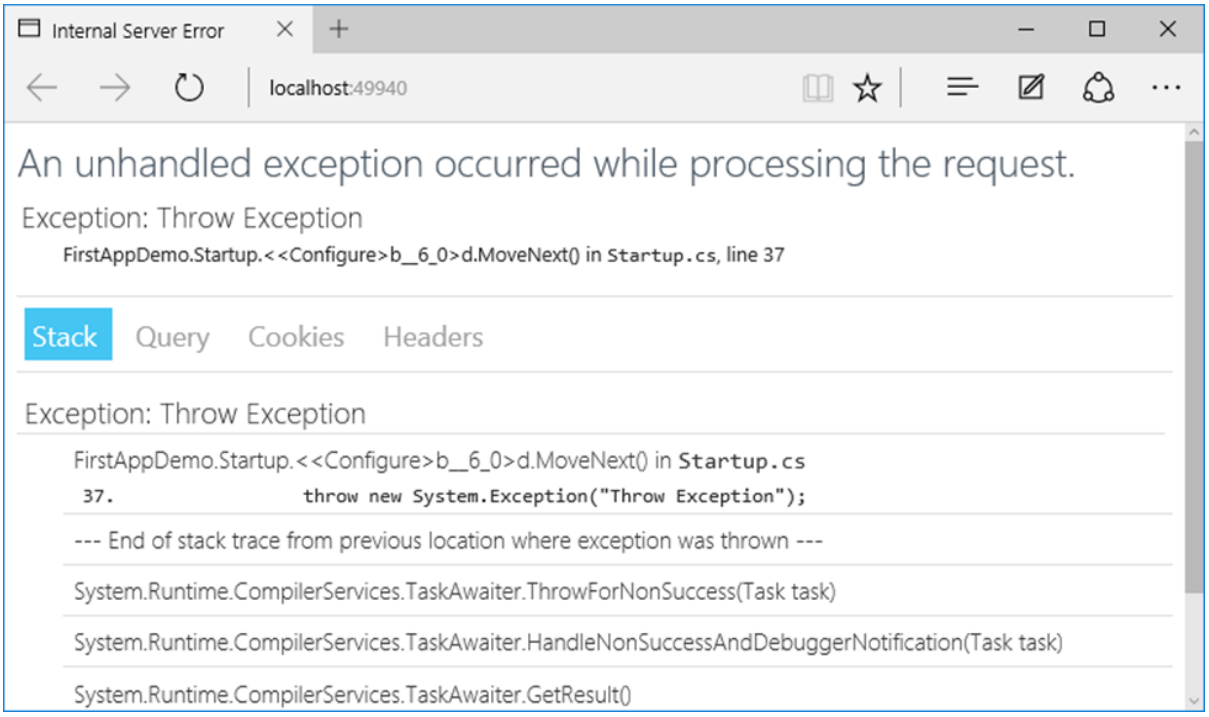
```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();

    app.UseDeveloperExceptionPage();
    app.UseRuntimeInfoPage();

    app.Run(async (context) =>
    {
        throw new System.Exception("Throw Exception");
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

- This piece of middleware is a little bit different than the other pieces of middleware, the other pieces of middleware are typically looking at the incoming request and making some decision about that request.

- The UseDeveloperExceptionPage doesn't care so much about the incoming requests as it does what happens later in the pipeline.

- It is going to just call into the next piece of middleware, but then it is going to wait to see if anything later in the pipeline generates an exception and if there is an exception, this piece of middleware will give you an error page with some additional information about that exception.

Let us now run the application again. It will produce an output as shown in the following screenshot.



Now you will see some information that you would expect if there was an error in development. You will also get a stack trace and you can see that there was an unhandled exception thrown on line 37 of Startup.cs.

You can also see raw exception details and all of this information can be very useful for a developer. In fact, we probably only want to show this information when a developer is running the application.

9. ASP.NET Core — Static Files

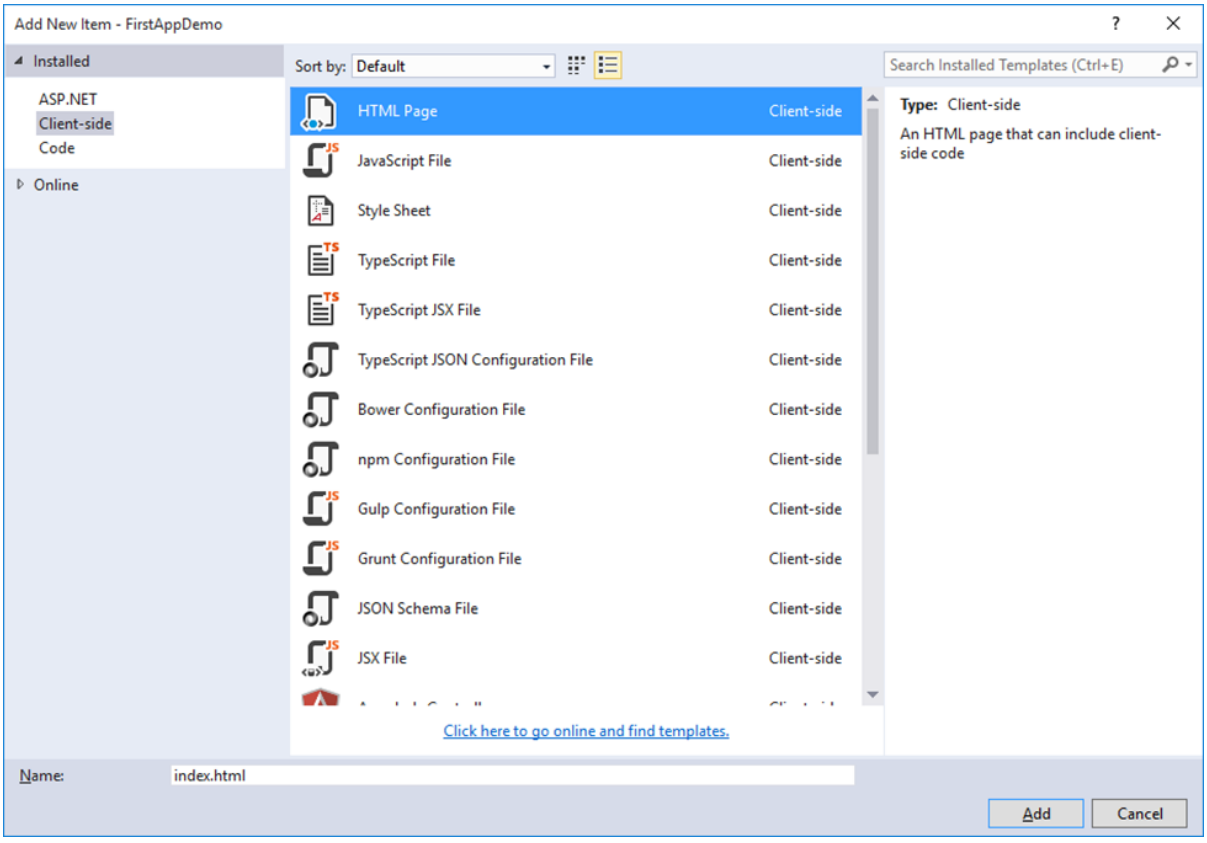
In this chapter, we will learn how to work with files. An important feature nearly every web application needs is the ability to serve up files (static files) from the file system.

- Static files like JavaScript files, images, CSS files that we have on the file system are the assets that ASP.NET Core application can serve directly to clients.
- Static files are typically located in the web root (wwwroot) folder.
- By default, that is the only place where we can serve up files directly from the file system.

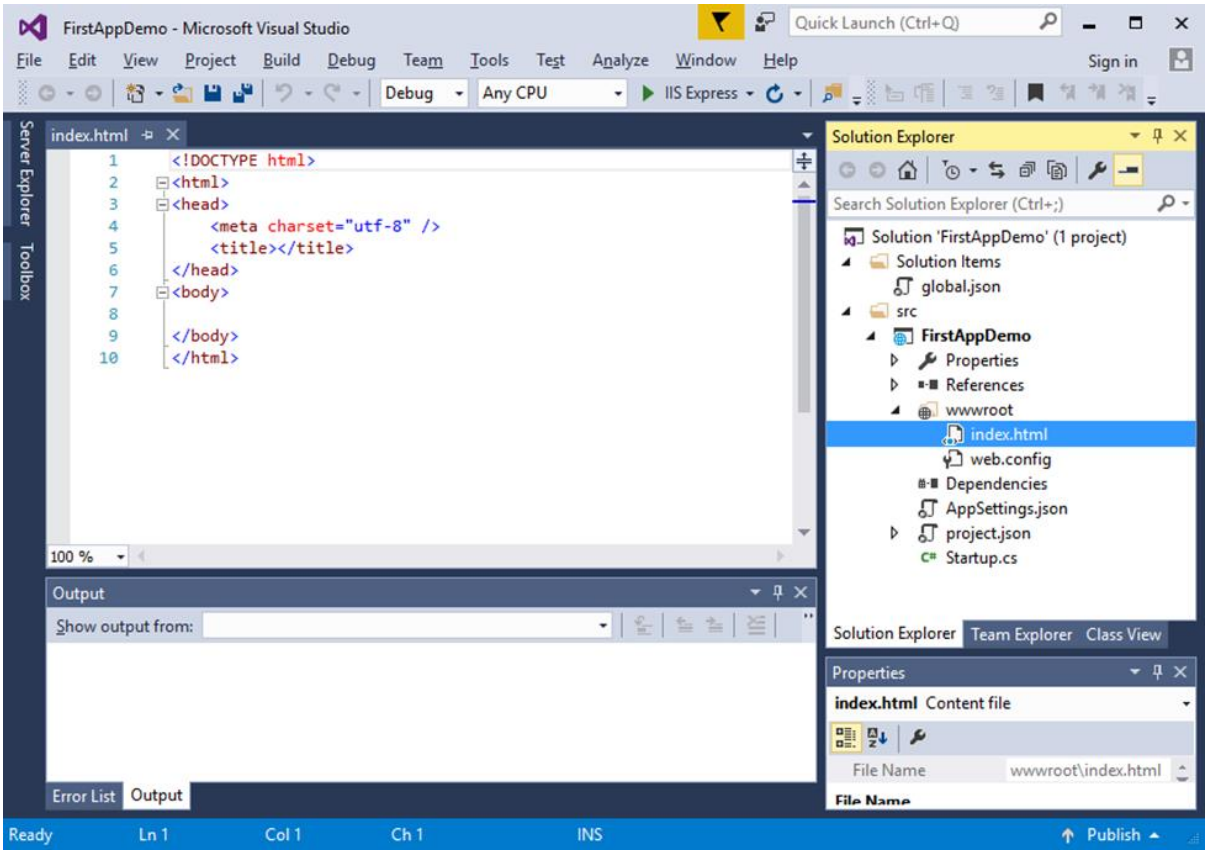
Example

Let us now take a simple example in which we will understand how we can serve those files in our application.

Here, we want to add a simple HTML file to our FirstAppDemo application and this HTML file has to go into the web root (wwwroot) folder. Right-click on wwwroot folder in the Solution Explorer and select **Add > New Item….**



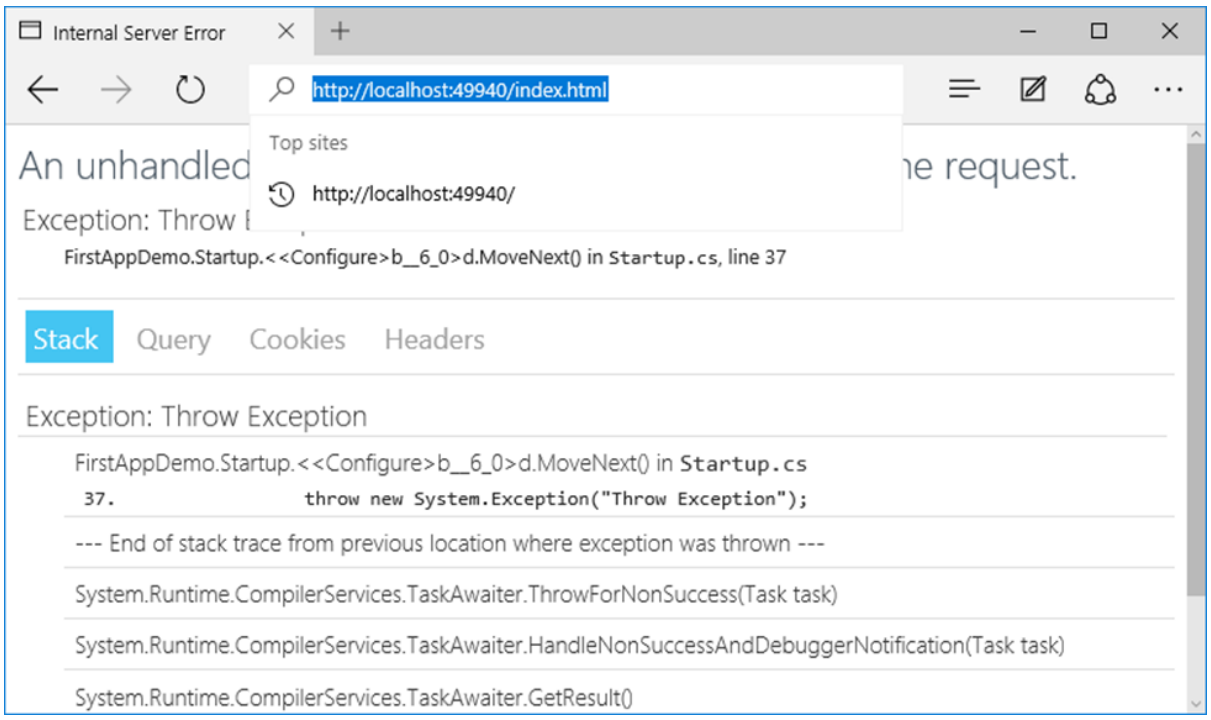
In the middle pane, select the **HTML Page** and call it **index.html** and click the **Add** button.



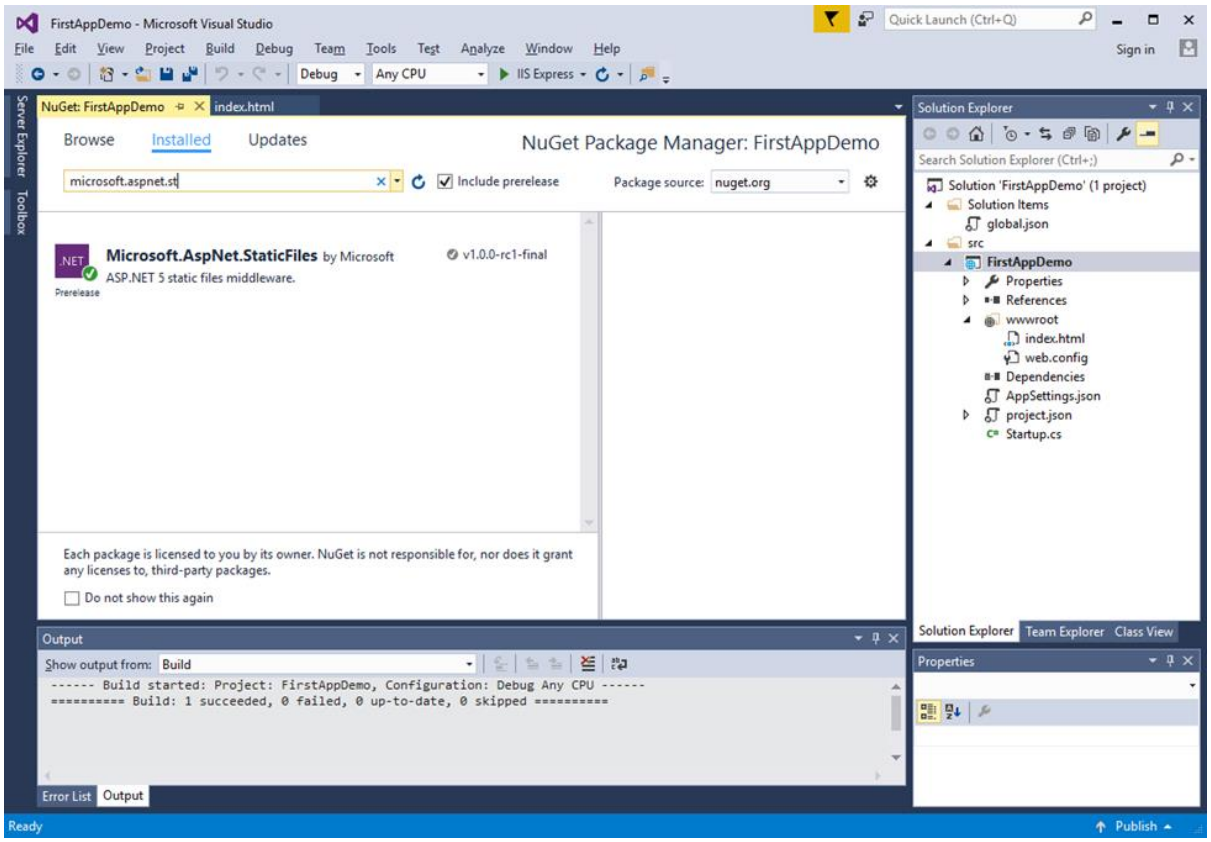
You will see a simple **index.html** file. Let us add some simple text and title as shown below.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Welcome to ASP.NET Core</title>
</head>
<body>
    Hello, Wolrd! this message is from our first static HTML file.
</body>
</html>
```

When you run your application and go to **index.html** in the browser, you will see that the **app.Run** middleware throws an exception because there is nothing currently in our application.



There is no piece of middleware that will go looking for any file on the file system to serve. To fix this issue, go to the **NuGet packages manager** by right-clicking on your project in Solution Explorer and selecting Manage NuGet Packages.



Search for **Microsoft.AspNet.StaticFiles** which will find the static files middleware. Let us install this nuget package and now we should have additional methods that we can use to register middleware inside the Configure method.

Let us add **UseStaticFiles** middle in Configure method as shown in the following program.

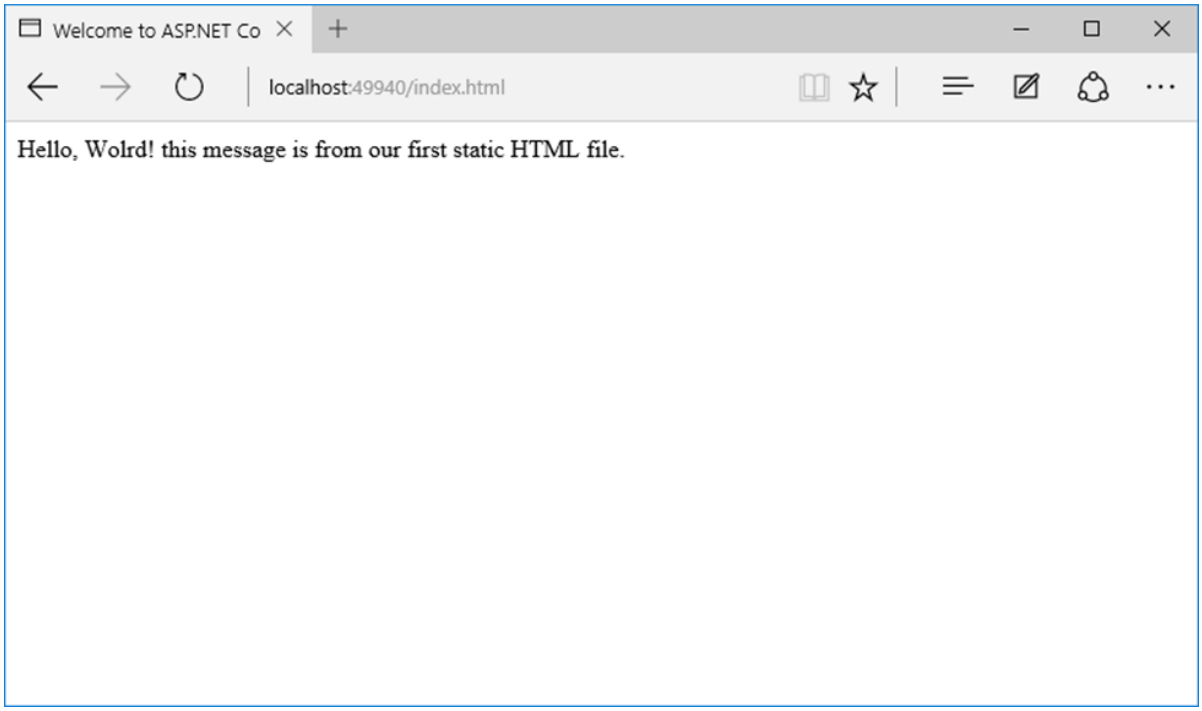
```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace FirstAppDemo
{
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.UseDeveloperExceptionPage();
            app.UseRuntimeInfoPage();
```

```
            app.UseStaticFiles();
            app.Run(async (context) =>
            {
                throw new System.Exception("Throw Exception");
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });
        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Unless you override the options and pass in some different configuration parameters, what static files will do is for a given request is to look at the **request path**. This request path is then compared to the file system and what is on the file system.

- If the static file sees a file that it can use, it will serve up that file and not call the next piece of middleware.
- If it doesn't find a matching file, then it will simply continue with the next piece of middleware.

Let us save the **Startup.cs** file and refresh your browser.



You can now see the index.html file. Anything that you put anywhere inside the wwwroot — any JavaScript file or CSS file or HTML file, you will be able to serve them up.

- Now if you want index.html to be your default file, this is a feature that IIS has always had.

- You can always give IIS a list of default files to look for. If someone came to the root of a directory or, in this case, the root of the website and if IIS found something named index.html, it would just automatically serve that file.

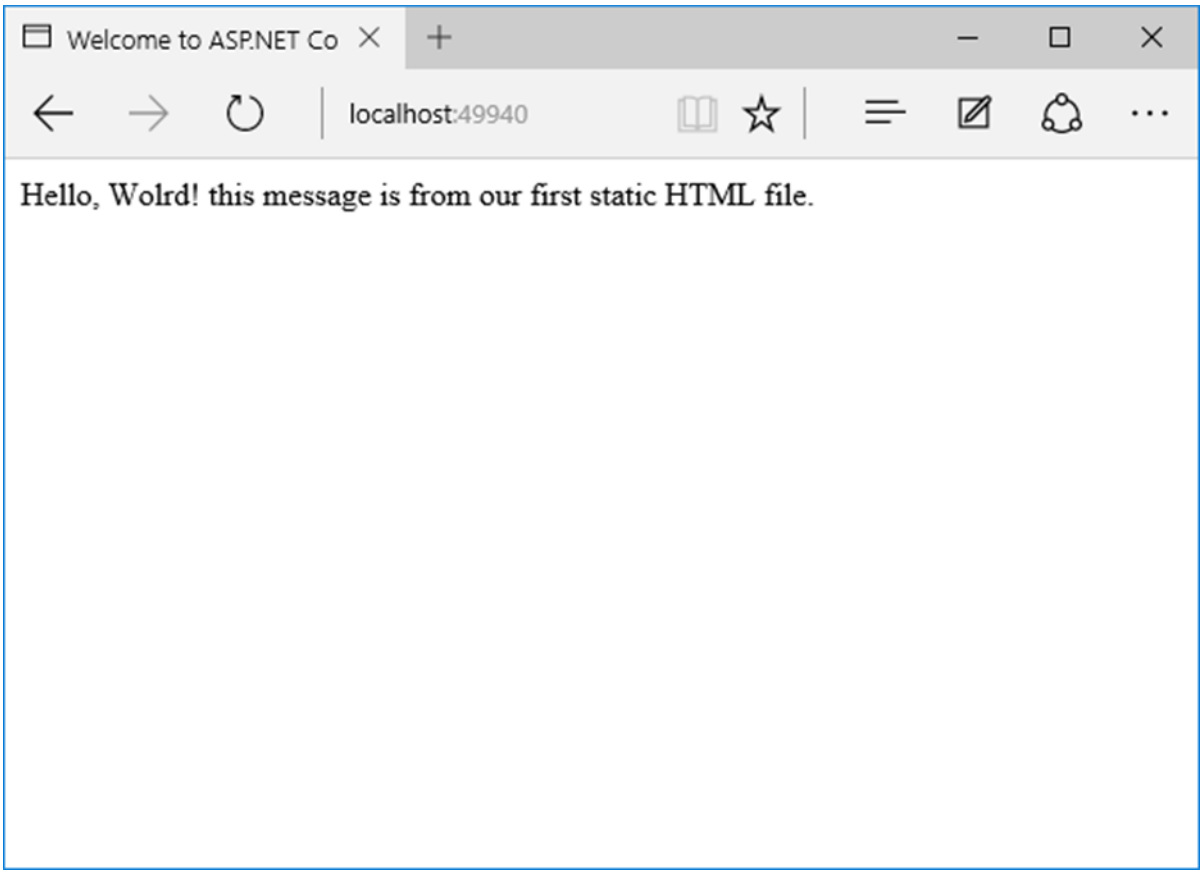
- Let us now start by making a few changes. First, we need to remove the forced error and then add another piece of middleware, which is UseDefaultFiles. The following is the implementation of the Configure method.

```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();

    app.UseDeveloperExceptionPage();
    app.UseRuntimeInfoPage();

    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

- This piece of middleware will look at an incoming request and see if it is for the root of a directory and if there are any matching default files.

- You can override the options for this piece of middleware to tell it what are the default files to look for, but Index.html is by default one of the default files.

Let us save the **Startup.cs** file and go to the root of the web application in your browser.



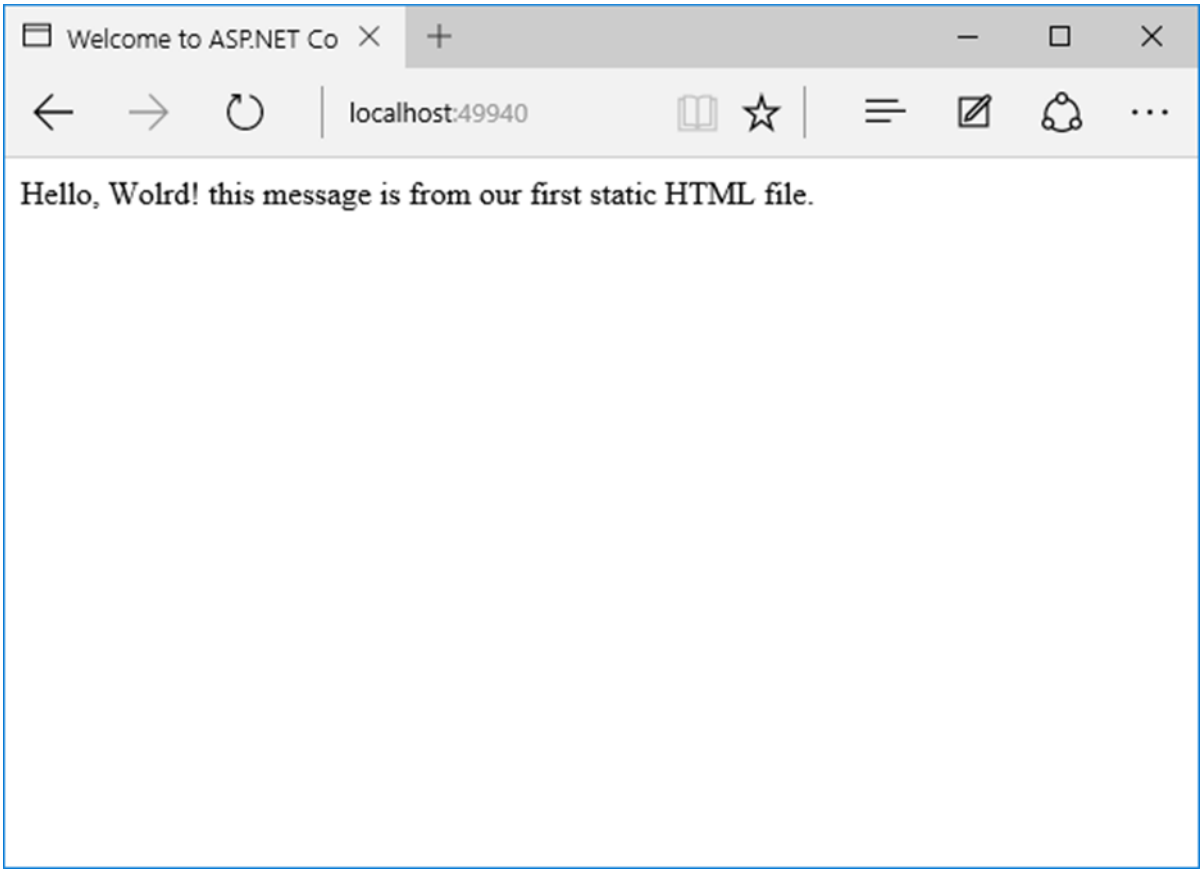
You can now see that the index.html is your default file. The order in which you install the middleware is important because if you had UseDefaultFiles after UseStaticFiles, you would not get the same result.

If you are going to use UseDefaultFiles and UseStaticFiles, you might also want another piece of middleware that is inside the Microsoft.aspnet.staticfiles, NuGet package, and that is the **FileServer middleware**. This essentially includes the Default Files and the Static Files in the correct order.

```
// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();

    app.UseDeveloperExceptionPage();
    app.UseRuntimeInfoPage();

    app. UseFileServer();

    app.Run(async (context) =>
```

```
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });
}
```

Let us save the **Startup.cs** file again. Once you refresh the browser, you will see the same result as shown in the following screenshot.



10. ASP.NET Core — Setup MVC

In this chapter, we will set up the MVC framework in our FirstAppDemo application. We will proceed by building a web application on top of the ASP.NET Core, and more specifically, the ASP.NET Core MVC framework. We can technically build an entire application using only middleware, but ASP.NET Core MVC gives us the features that we can use to easily create HTML pages and HTTP-based APIs.

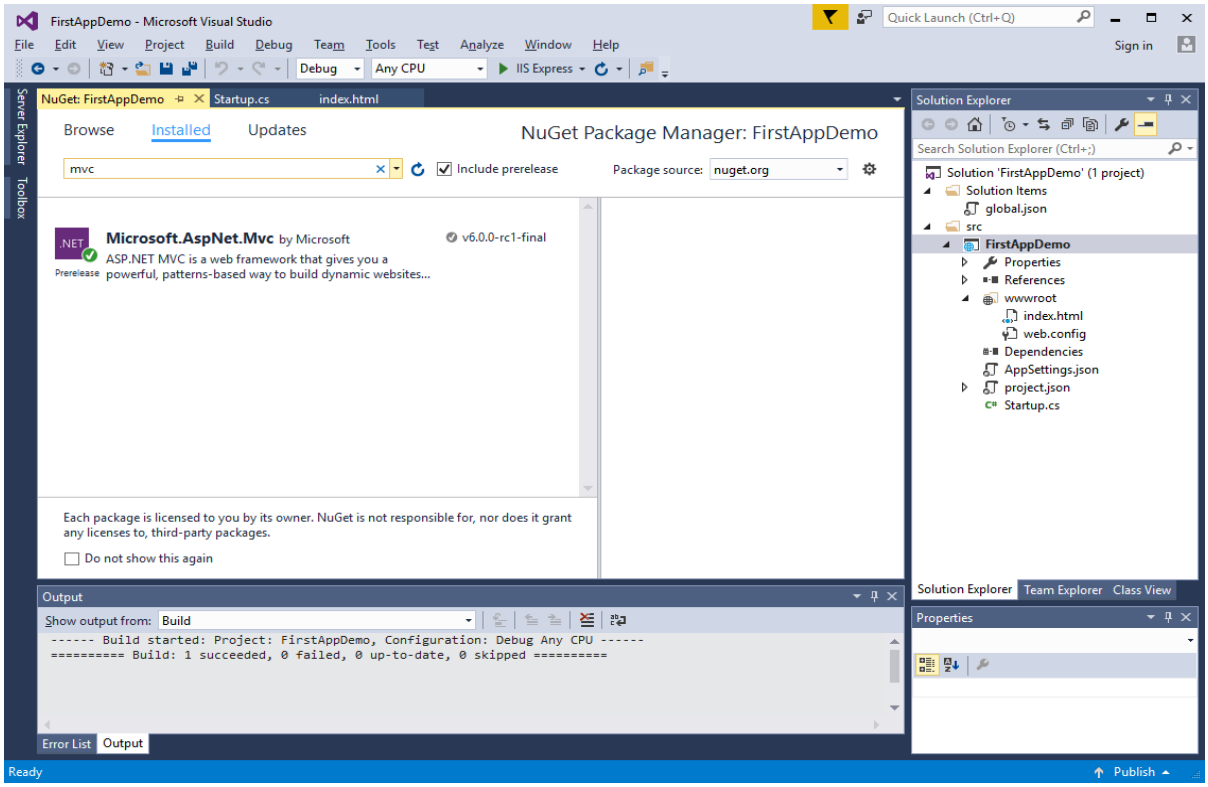
To setup MVC framework in our empty project, follow these steps:

- Install the **Microsoft.AspNet.Mvc** package, which gives us access to the assemblies and classes provided by the framework.

- Once the package is installed, we need to register all of the services that ASP.NET MVC requires at runtime. We will do this inside the **ConfigureServices** method.

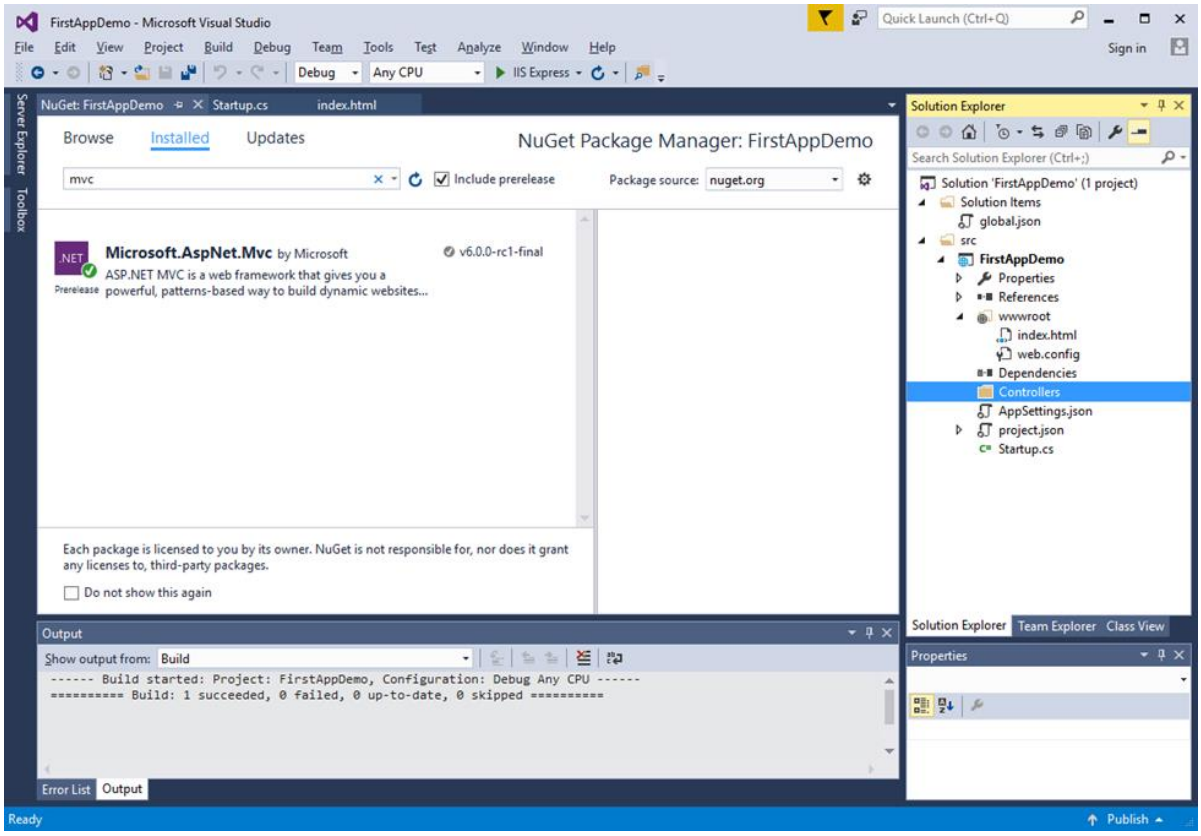
- Finally, we need to add middleware for ASP.NET MVC to receive requests. Essentially this piece of middleware takes an HTTP request and tries to direct that request to a C# class that we will write.

Step 1: Let us go to the NuGet package manager by right-clicking on the Manage NuGet Packages. Install the Microsoft.AspNet.Mvc package, which gives us access to the assemblies and classes provided by the framework.

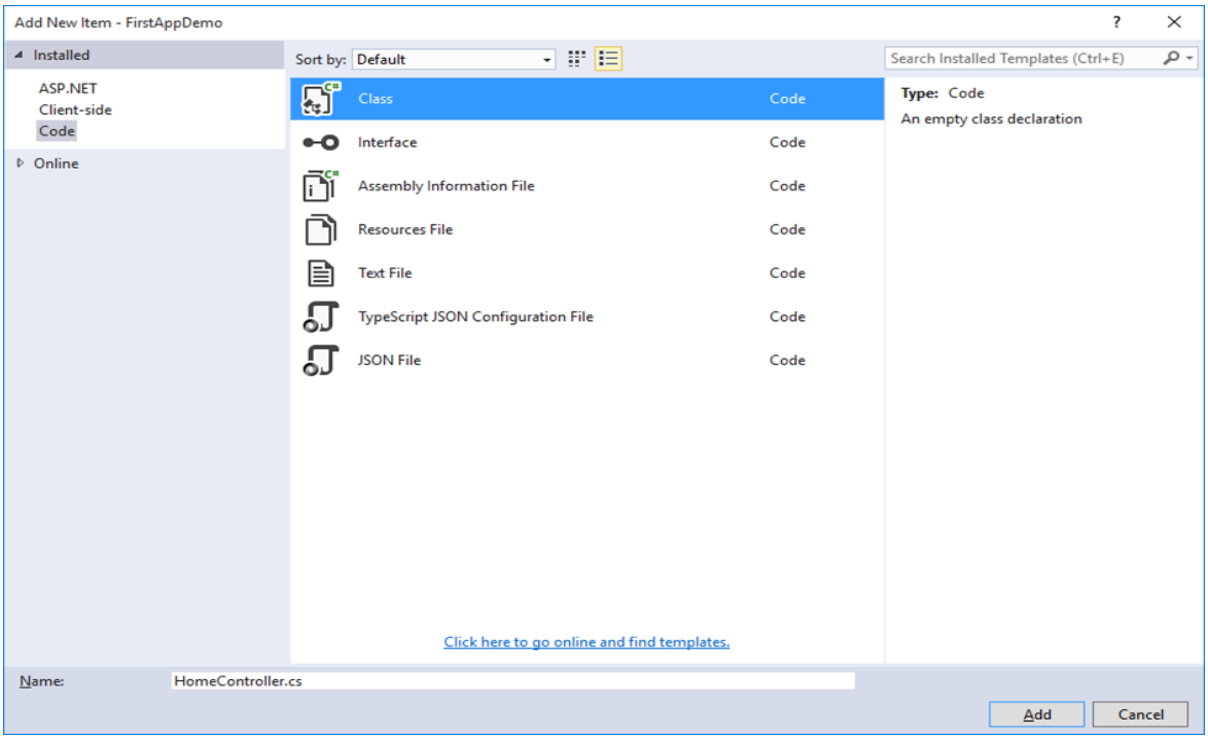


Step 2: Once the Microsoft.AspNet.Mvc package is installed, we need to register all the services that ASP.NET Core MVC requires at runtime. We will do this with the ConfigureServices method. We will also add a simple controller and we will see some output from that controller.

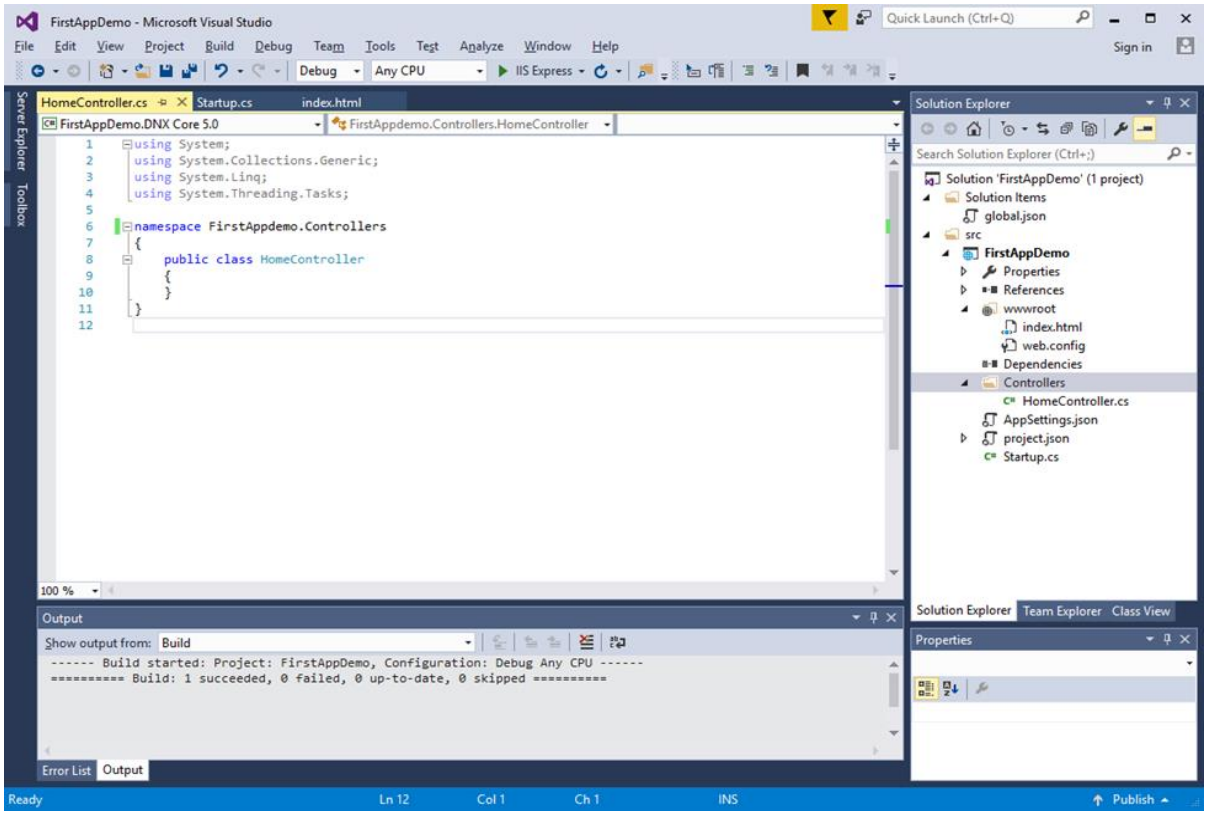
Let us add a new folder to this project and call it **Controllers.** In this folder, we can place multiple controllers as shown below in the Solution Explorer.



Now right-click on the Controllers folder and select on the **Add > Class** menu option.



Step 3: Here we want to add a simple **C#** class, and call this class **HomeController** and then Click on the Add button as in the above screenshot.

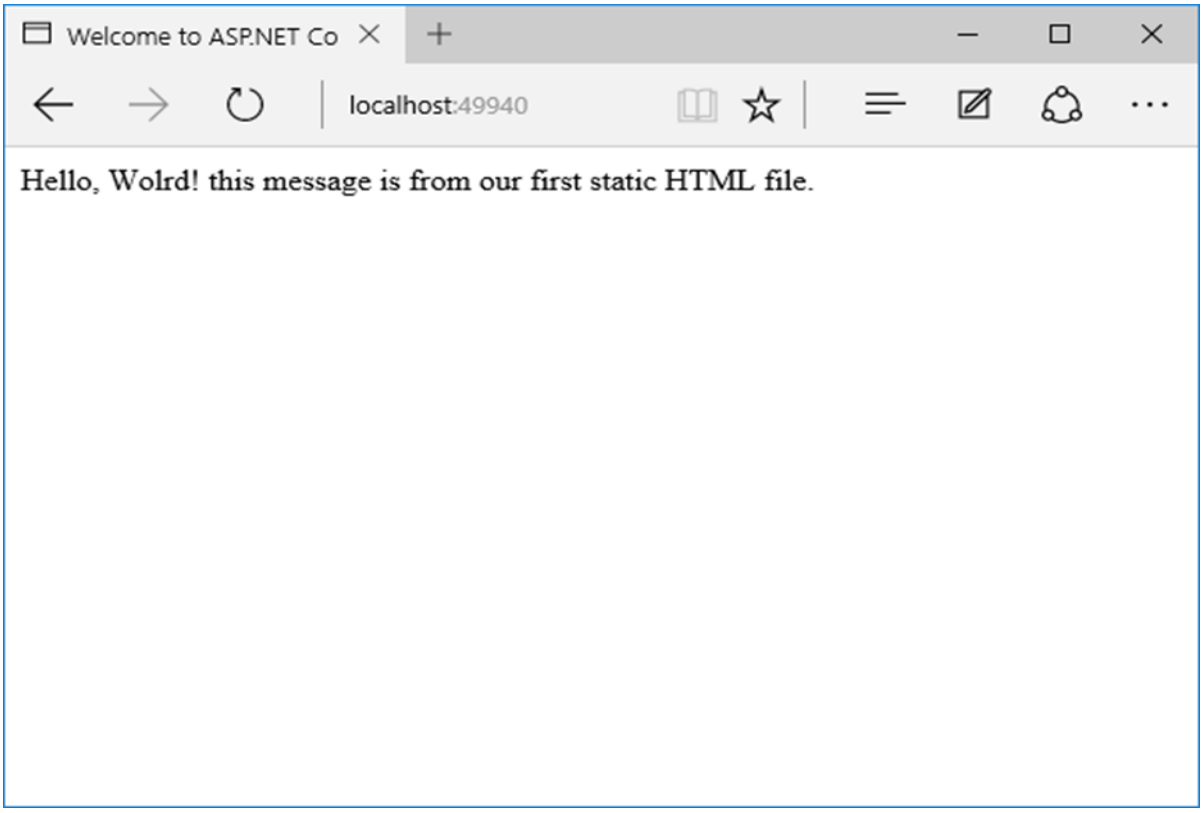


This will be our default page.

Step 4: Let us define a single public method that returns a string and call that method Index as shown in the following program.

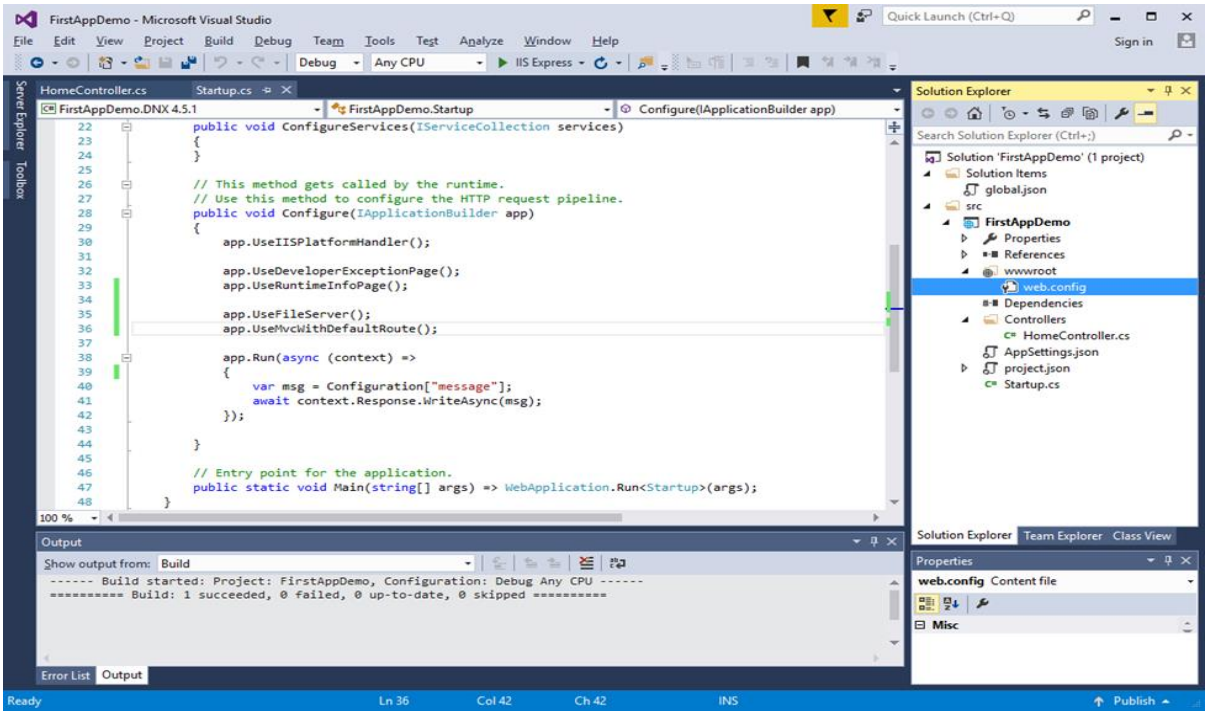
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
{
    public class HomeController
    {
        public string Index()
        {
            return "Hello, World! this message is from Home Controller...";
        }
    }
}
```

Step 5: When you go to the root of the website, you want to see the controller response. As of now, we will be serving our index.html file.

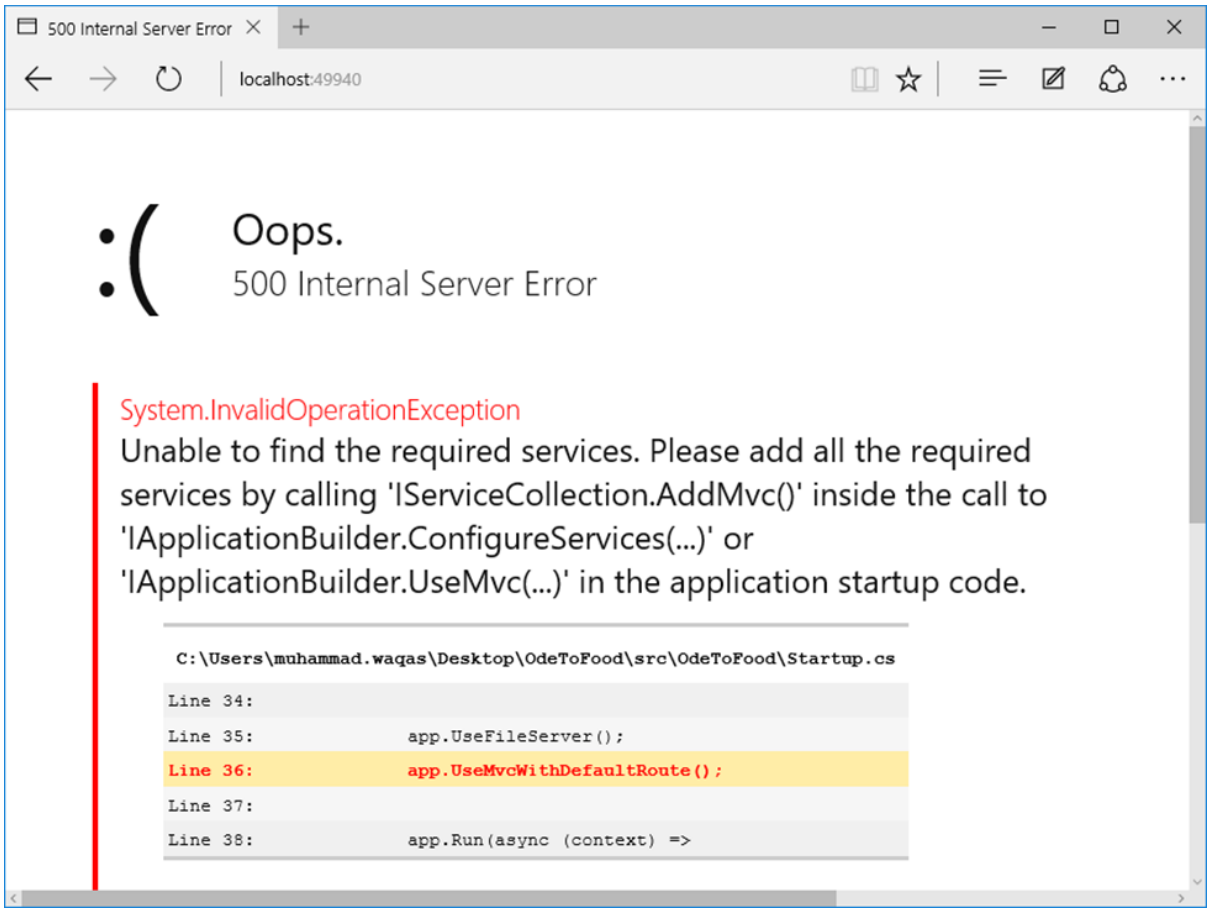


Let us go into the root of the website and delete **index.html**. We want the controller to respond instead of index.html file.

Step 6: Now go to the Configure method in the Startup class and add the **UseMvcWithDefaultRoute** piece of middleware.



Step 7: Now refresh the application at the root of the website.



You will encounter a 500 error. The error says that the framework was unable to find the required ASP.NET Core MVC services.

The ASP.NET Core Framework itself is made up of different small components that have very focused responsibilities.

For example, there is a component that has to locate and instantiate the controller.

That component needs to be in the service collection for ASP.NET Core MVC to function correctly.

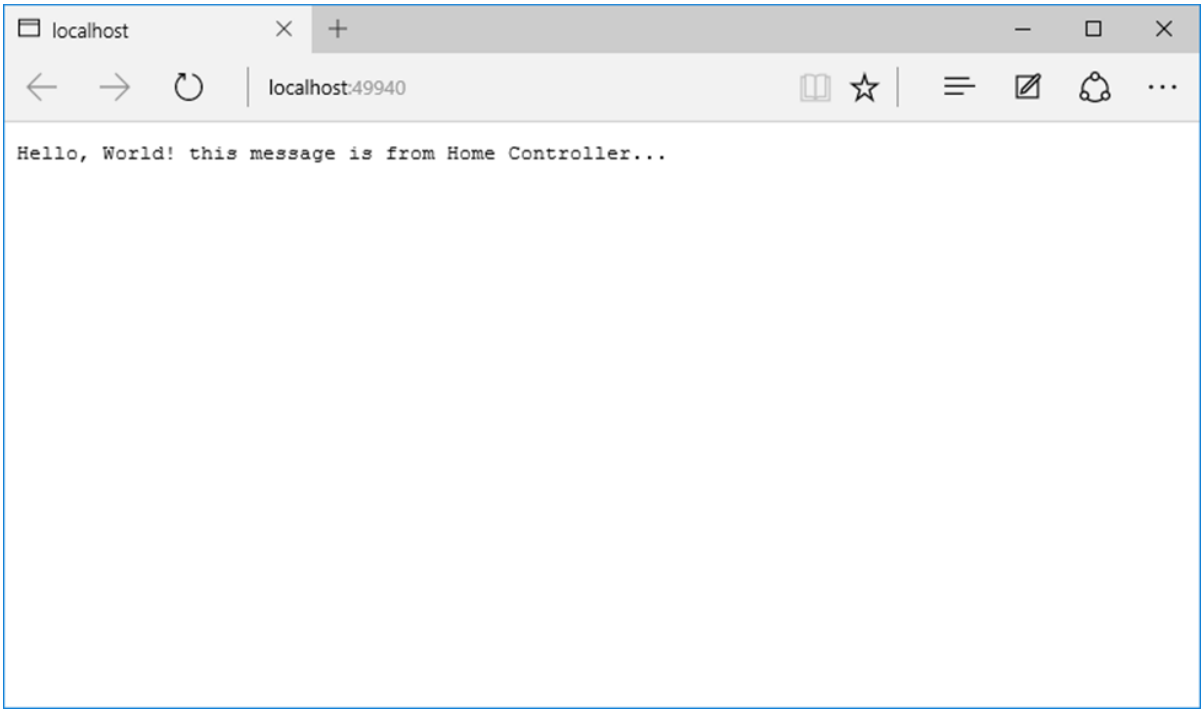
Step 8: In addition to adding the NuGet package and the middleware, we also need to add the AddMvc service in the ConfigureServices. Here is the complete implementation of the Startup class.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace FirstAppDemo
{
```

```
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.UseDeveloperExceptionPage();
            app.UseRuntimeInfoPage();

            app.UseFileServer();
            app.UseMvcWithDefaultRoute();

            app.Run(async (context) =>
            {
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });
        }
        // Entry point for the application.
```

```
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Step 9: Save the **Startup.cs** file and go to the browser and refresh it. You will now receive a response from our **home controller**.



11. ASP.NET Core — MVC Design Pattern

The MVC (Model-View-Controller) design pattern is a design pattern that's actually been around for a few decades, and it's been used across many different technologies, everything from Smalltalk to C++ to Java and now in C# and .NET as a design pattern to use when you're building a user interface.

- The MVC design pattern is a popular design pattern for the user interface layer of a software application.
- In larger applications, you typically combine a model-view-controller UI layer with other design patterns in the application, like data access patterns and messaging patterns.
- These will all go together to build the full application stack

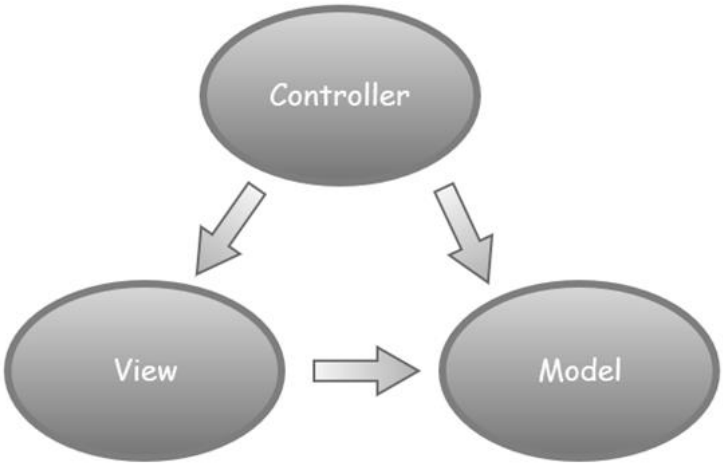
The MVC separates the user interface (UI) of an application into the following three parts:

- **The Model**: A set of classes that describes the data you are working with as well as the business logic.
- **The View**: Defines how the application's UI will be displayed. It is a pure HTML which decides how the UI is going to look like.
- **The Controller**: A set of classes that handles communication from the user, overall application flow, and application-specific logic

Idea behind MVC

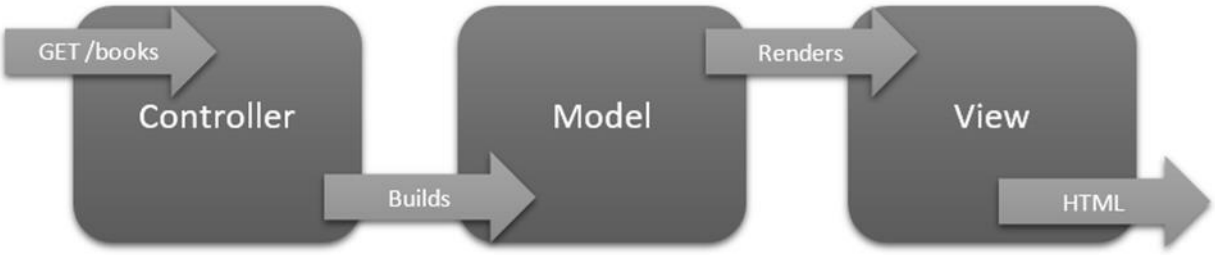
Let us now understand the idea behind MVC.

- The idea is that you'll have a component called the view which is solely responsible for rendering this user interface whether it should be HTML or whether it actually should be a UI widget on a desktop application.
- The view talks to a model, and that model contains all the data that the view needs to display.
- In a web application, the view might not have any code associated with it at all.
- It might just have HTML and then some expressions of where to take the pieces of data from the model and plug them into the correct places inside the HTML template that you've built in the view.



- The controller organizes everything. When an HTTP request arrives for an MVC application, the request gets routed to a controller, and then it's up to the controller to talk to either the database, the file system, or a model.

In MVC, the controller receives an HTTP request, the controller has to figure out how to put together the information to respond to this request. Perhaps the user is directing the browser to the /books URL of the application. So the controller needs to put together the information to display a list of books. In this scenario, the controller will build a model.



- The model doesn't know anything about the HTTP request or the controller.
- The model is only responsible for holding the books information that the user wants to see, as well as any logic associated with that list of books.
- The model is just another C# class we can use and you might have more than one class if you have a complex model.
- Once the model is put together, the controller can then select a view to render the model.
- The view will take the information in the model, like all the books and each book title etc., and it will use that information to construct an HTML page.
- Then that HTML is sent back to the client in the HTTP response and the entire HTTP request and response transaction is completed.

These are the basics of the MVC design pattern and the idea behind this pattern is to keep a separation of concerns. So the controller is only responsible for taking a request and building a model. It is the model that carries the logic and data we need into the view. Then the view is only responsible for transforming that model into HTML.

12. ASP.NET Core — Routing

In MVC framework, we have three components, each with its own focus on a specific part of the job. In order for all of this to work, we need to find a way to send these HTTP requests to the right controller. In ASP.NET Core MVC, this process is known as routing. Routing is the process of directing an HTTP request to a controller.

Let us now understand how to route requests to different controllers.

- The ASP.NET Core middleware needs a way to determine if a given HTTP request should go to a controller for processing or not.
- The MVC middleware will make this decision based on the URL and some configuration information we provide. In this chapter, we will define this configuration information or you can say routing information inside Startup.cs when we add the MVC middleware.
- This approach is often referred to as the convention-based routing. The following is a code snippet for conventional routing.

```
routeBuilder.MapRoute("Default",
                "{controller=Home}/{action=Index}/{id?}");
```

- In this approach, we define templates that tell MVC how to look at a URL and find a controller name and an action name where a controller is a C# class and an action is a public method on that class.

In the last chapter, we have created a controller (HomeController) in our application which is a C# class and doesn't need to be derived from a base class or implement an interface or have any special attribute. It is a plain C# class with a name, HomeController and it contains the Index method which returns a string.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
{
    public class HomeController
    {
        public string Index()
        {
            return "Hello, World! this message is from Home Controller...";
        }
    }
```

```
}
```

Here, we are going to focus on **routing to controllers**. We will also try understanding how routing works.

Let us now return to the **Startup class** where we configured MVC middleware into our application. Inside the Configure method, we have used a method **UseMvcWithDefaultRoute**.

```
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();

    app.UseDeveloperExceptionPage();
    app.UseRuntimeInfoPage();

    app.UseFileServer();
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

This gives us a default routing rule that allows us to get to the **HomeController**. Instead of using the **UseMvcWithDefaultRoute**, let us use the **UseMvc**, and then configure the route at this point using a named method **ConfigureRoute**. The following is the implementation of the Startup.cs file.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNet.Routing;
using System;

namespace FirstAppDemo
{
```

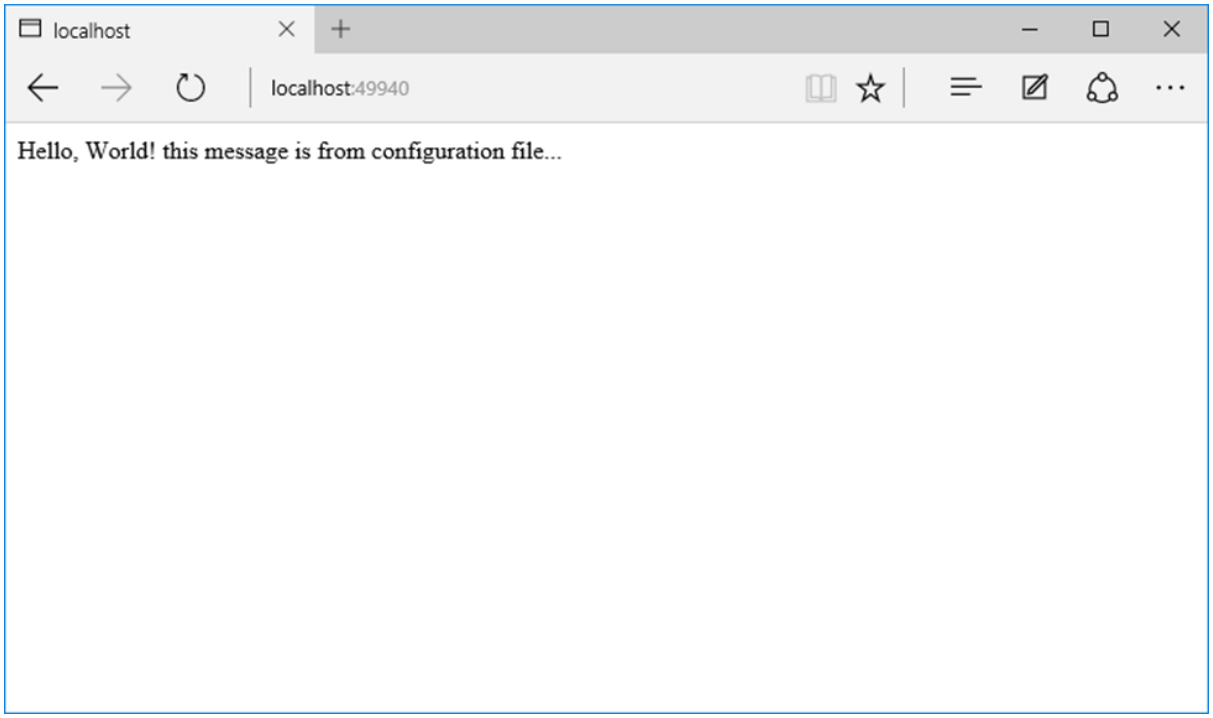
```
    public class Startup
    {
        public Startup()
        {
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.UseDeveloperExceptionPage();
            app.UseRuntimeInfoPage();

            app.UseFileServer();
            app.UseMvc(ConfigureRoute);

            app.Run(async (context) =>
            {
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });

        }
```

```
        private void ConfigureRoute(IRouteBuilder routeBuilder)
        {
            //Home/Index
            routeBuilder.MapRoute("Default",
"{controller=Home}/{action=Index}/{id?}");
        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Inside the **ConfigureRoute** method, you can configure your routes; you can see that this method has to take a parameter of type IRouteBuilder. The goal of routing is to describe the rules that ASP.NET Core MVC will use to process an HTTP request and find a controller that can respond to that request.

- You can have one route that can map requests to different controllers.
- We can tell the routeBuilder that we want to map a new route and its name is "Default" and then provide the most important piece of routing information, which is the template.
- The template is a string, and it is going to describe to ASP.NET Core MVC how to pull apart a URL.
- In the last example, we have added a HomeController, so you can also request any of the following URLs, and they will be directed to the Index action on the HomeController as well
    - http://localhost:49940
    - http://localhost:49940/Home
    - http://localhost:49940/Home/Index
- When a browser requests **http://mysite/**or **http://mysite/Home**, it gets back the output from the HomeController's Index method.
- You can try this as well by changing the URL in the browser. In this example, it is **http://localhost:49940/**, except that the port which might be different.
- If you append /Home or /Home/Index to the URL and press the Enter button, you will see the same result.
- The question mark at the end of ID means that this parameter is optional. In other words, ASP.NET Core MVC doesn't have to see some sort of ID here, which might be a number, or a string or GUID.

Let us run the application in the browser. Once the application is run, you will see the following output.



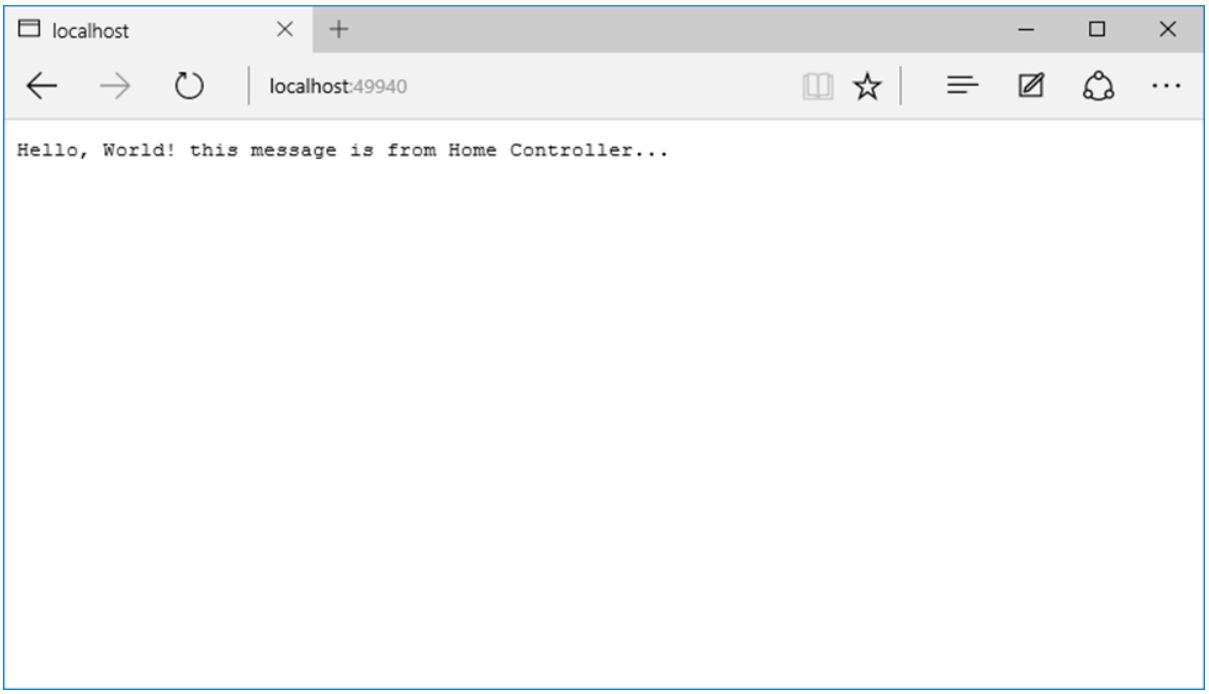
You can see a message pop up from the app.Run middleware, and the reason we are getting this message is because the MVC middleware saw that URL. This was a request to the root of the website, which didn't find a controller name or an action name in the URL. The root of the website gave up processing that request and passed the request to the next piece of middleware, which is the **app.Run** code. The template for the route we have specified is quiet unlike the default template.

In the default template, there are some default values to apply if a controller and an action name aren't found. If a request comes in to the root of the website, the default controller name would be Home. You could change that to any other controller as you want and the default action name can be Index. You can also change the default action if you want as shown in the following program.

```
private void ConfigureRoute(IRouteBuilder routeBuilder)
{
     //Home/Index
     routeBuilder.MapRoute("Default",
"{controller=Home}/{action=Index}/{id?}");
}
```

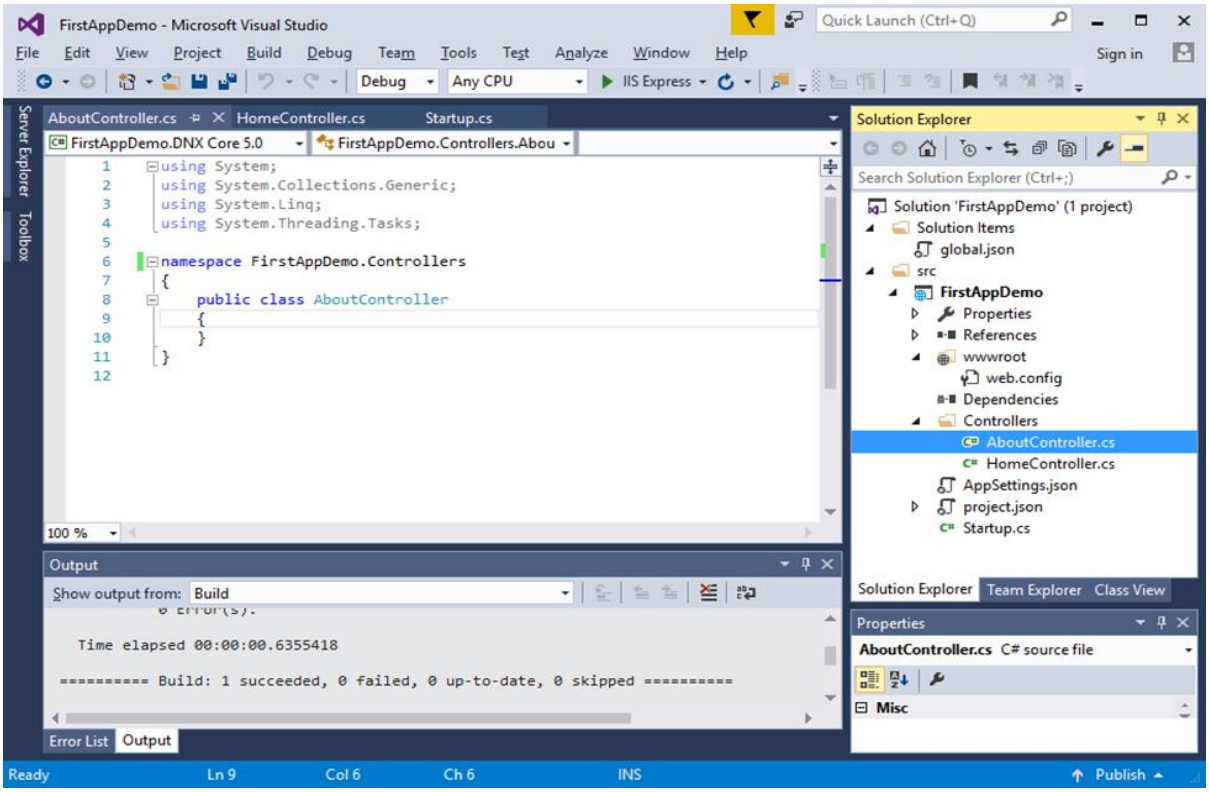
If a request comes in to the root of the website, MVC doesn't see a controller/action type of URL, but it can use these defaults.

Let us save the Startup.cs file and refresh the browser to the root of the website.



You will now see a response from your controller and you can also go to **/home** which will invoke the default action, which is index. You can also go to /home/index and now MVC will pulling the controller name and action name out of the URL.

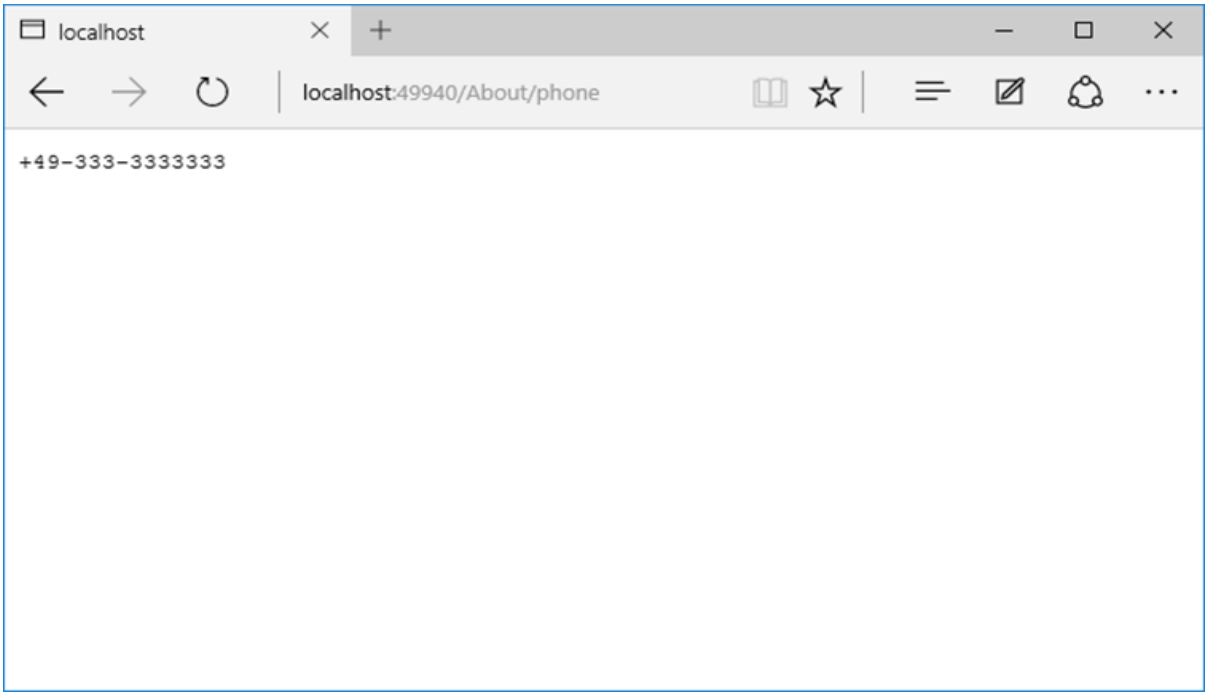
Let us create another controller by adding another class and call it **AboutController**.



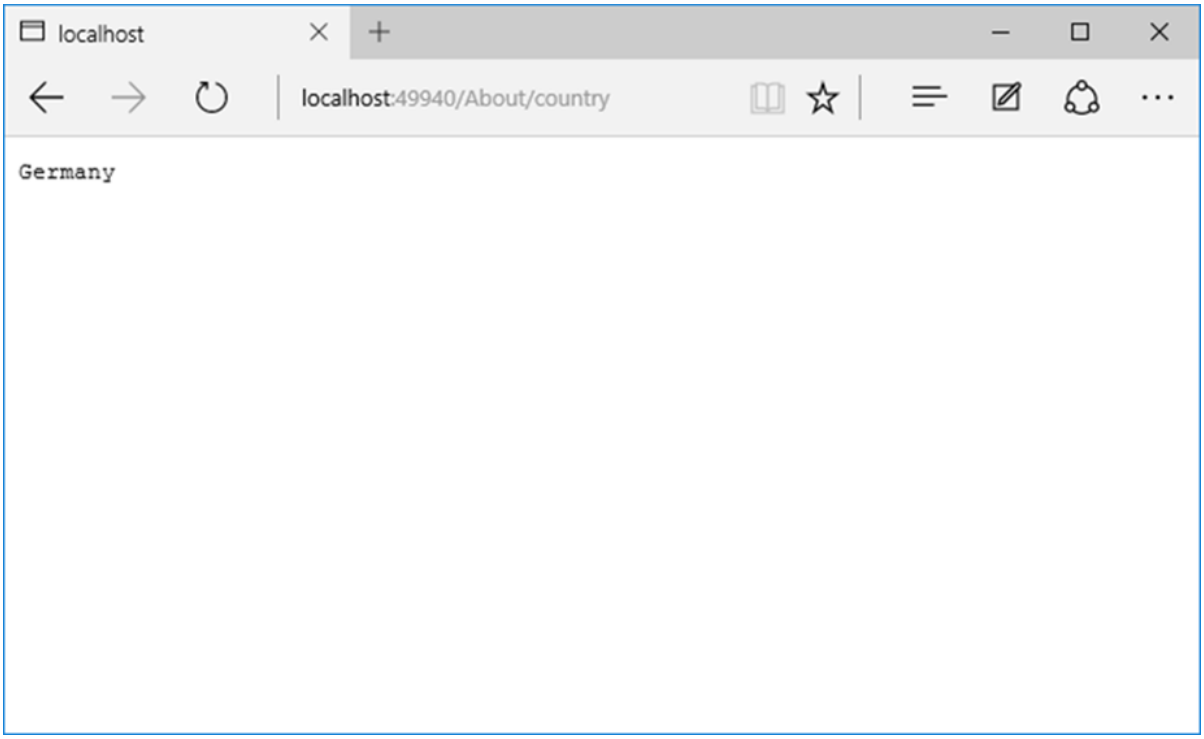
Let us add some simple action methods which will return string as shown in the following program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Controllers
{
    public class AboutController
    {
        public string Phone()
        {
            return "+49-333-3333333";
        }

        public string Country()
        {
            return "Germany";
        }
    }
}
```

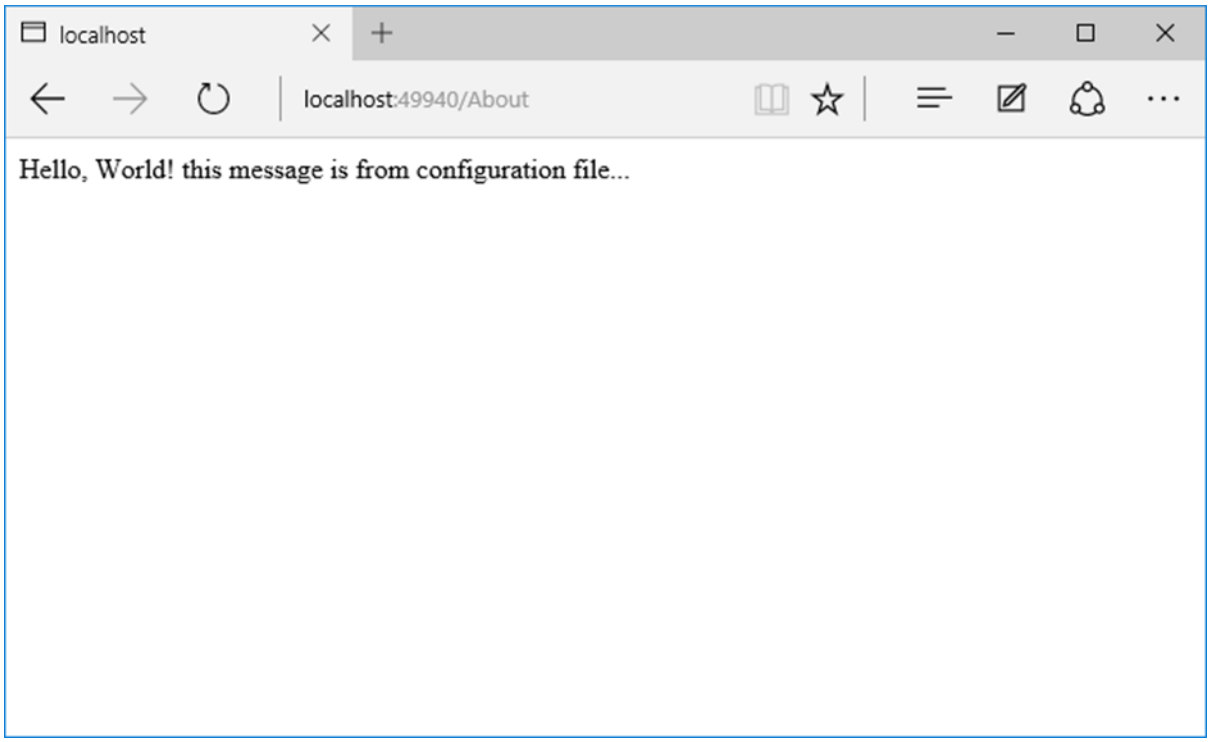
In this controller, you can see two actions methods — Phone and Country, which will return just a phone number and country name respectively. We'll get into fancy HTML later. Let us save this file and specify /about/phone at the end of the root URL.



You can see the phone number as in the above screenshot. If you specify **/about/country**, you will see the name of the country too.



If you go to **/about**, it is again going to fall through the middleware and go to your app.Run middleware and you will see the following page.



Here, the ASP.NET Core MVC goes to the AboutController, but does not find an action specified. So it will default to Index and this controller doesn't have an Index method and then the request will go the next piece of middleware.

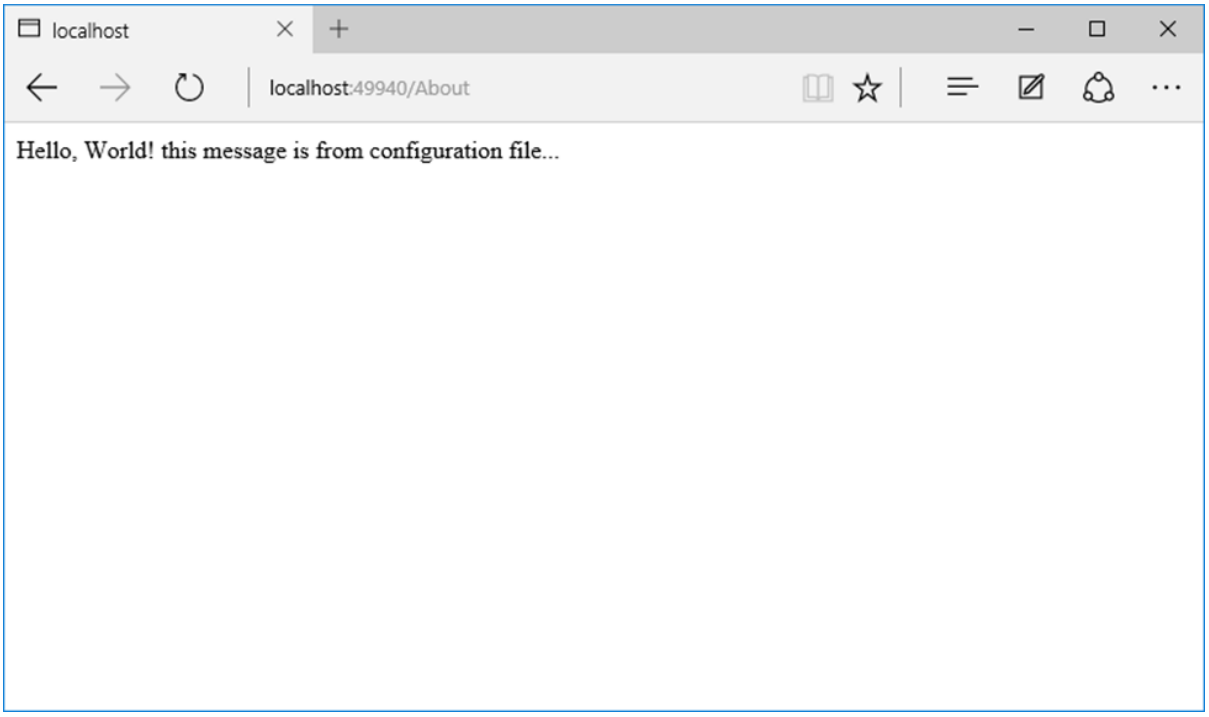
13. ASP.NET Core — Attribute Routes

In this chapter, we will learn another approach to routing and that is attribute-based routing. With attribute-based routing, we can use C# attributes on our controller classes and on the methods internally in these classes. These attributes have metadata that tell ASP.NET Core when to call a specific controller.

- It is an alternative to convention-based routing.
- Routes are evaluated in the order that they appear, the order that you register them in, but it's quite common to map multiple routes particularly if you want to have different parameters in the URL or if you want to have different literals in the URL.

Example

Let us take a simple example. Open the **FirstAppDemo** project and run the application in the browser. When you specify **/about,** it will produce the following output:

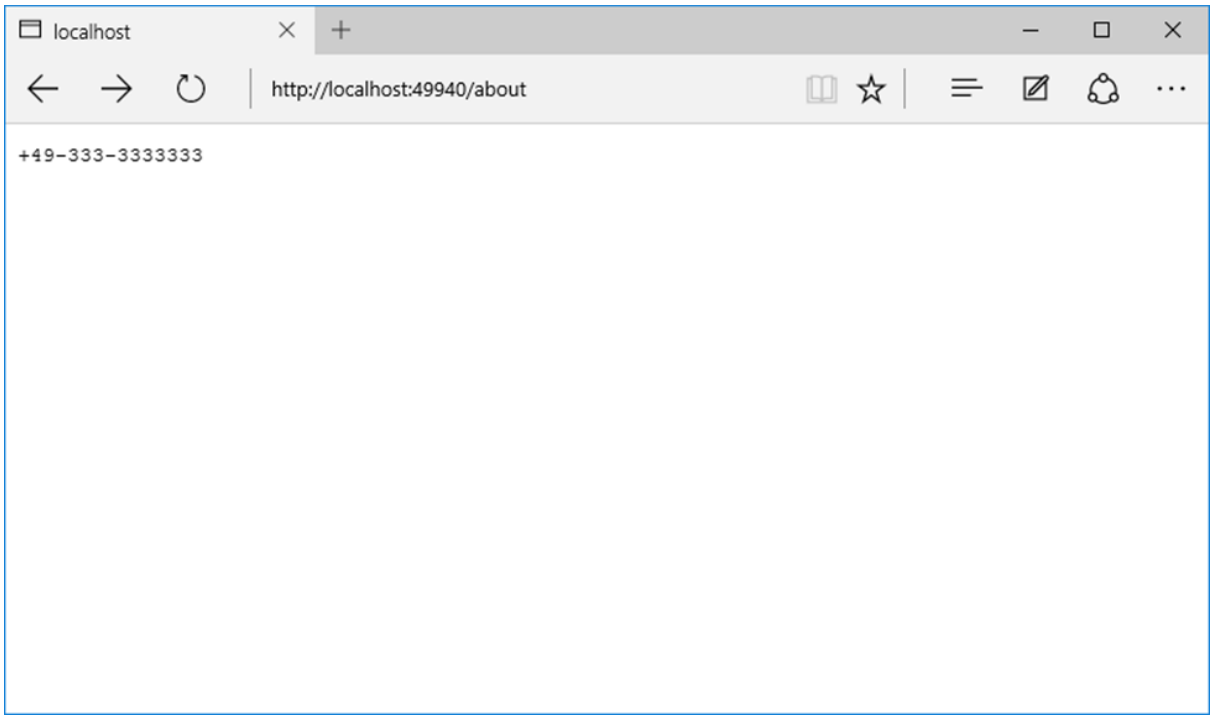


What we want here is that when we specify **/about**, the application should invoke the Phone action of the AboutController. Here, we can enforce some explicit routes for this controller using a Route attribute. This attribute is in the namespace **Microsoft.AspNet.Mvc**.

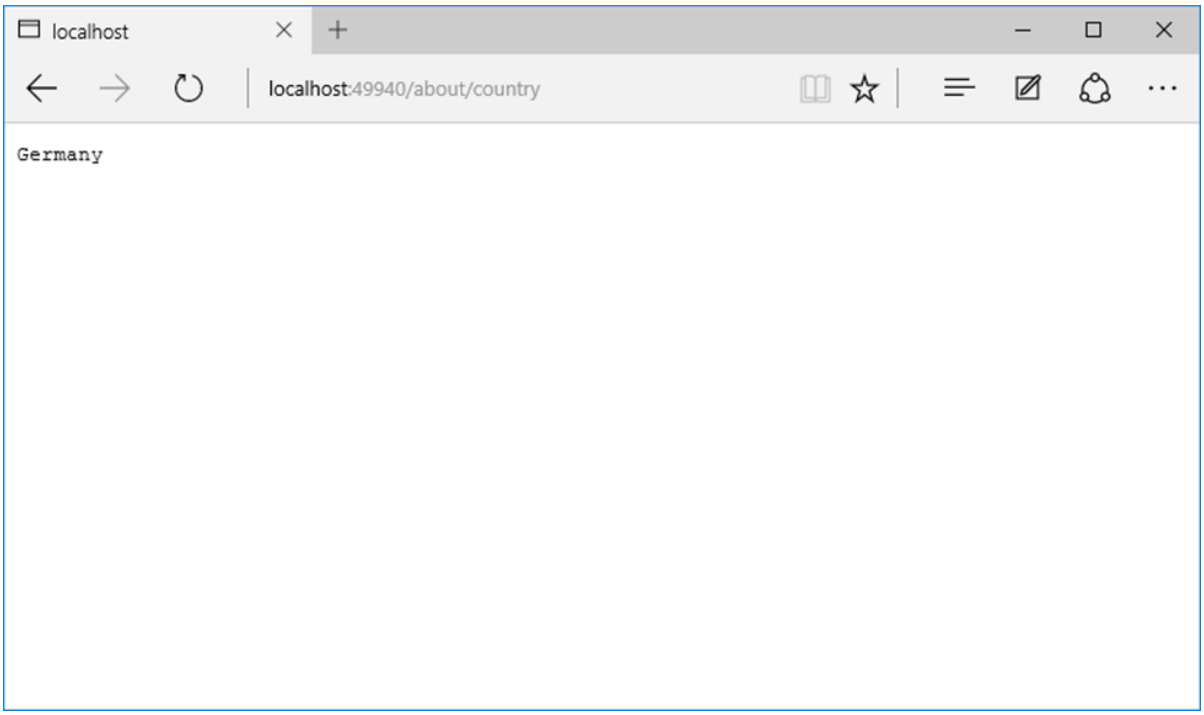
The following is the implementation of **AboutController** in which the attribute routes are added.

```
using Microsoft.AspNet.Mvc;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Controllers
{
    [Route("about")]
    public class AboutController
    {
        [Route ("")]
        public string Phone()
        {
            return "+49-333-3333333";
        }

        [Route("country")]
        public string Country()
        {
            return "Germany";
        }
    }
}
```

Here we want this route to look like about and for the Phone action we have specified an empty string, which means that we don't need the action to be specified to get this method. The user just needs to come to /about. For the Country action we have specified the "country" in the route attribute. Let us save the AboutController, refresh your browser and go to the /about and should give you the Phone action.



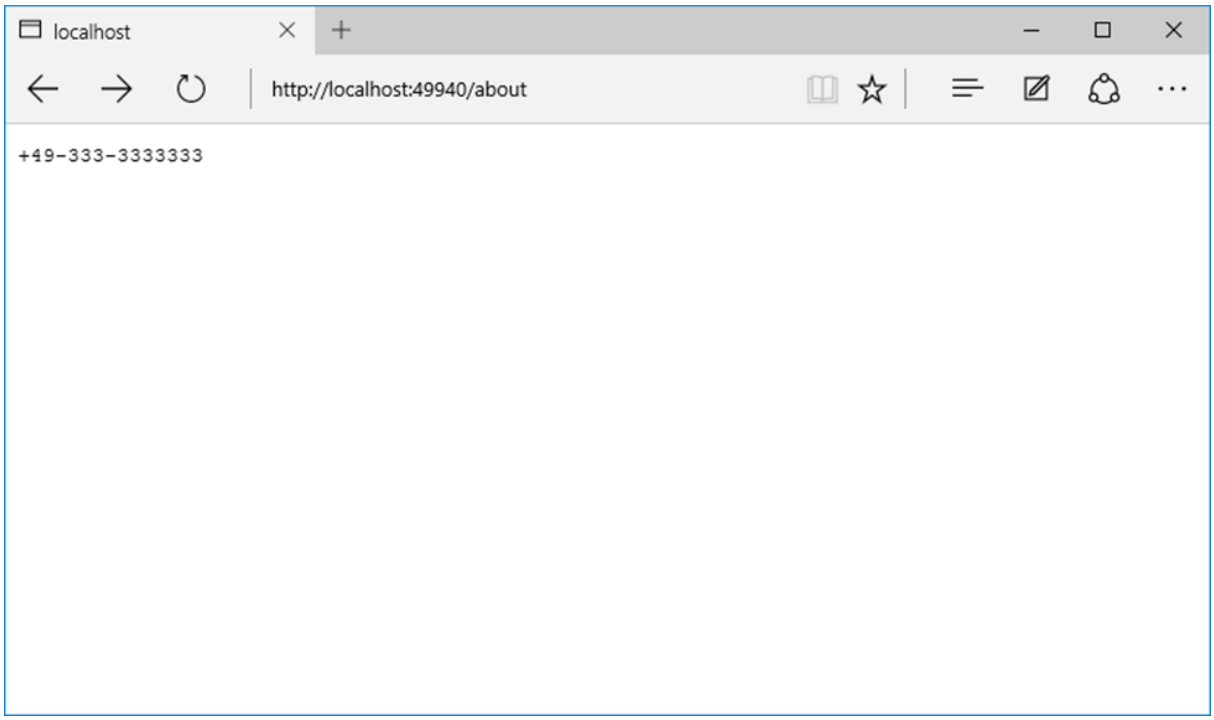
Let us now specify **/about/country**. This will still allow you to get to that Country action.



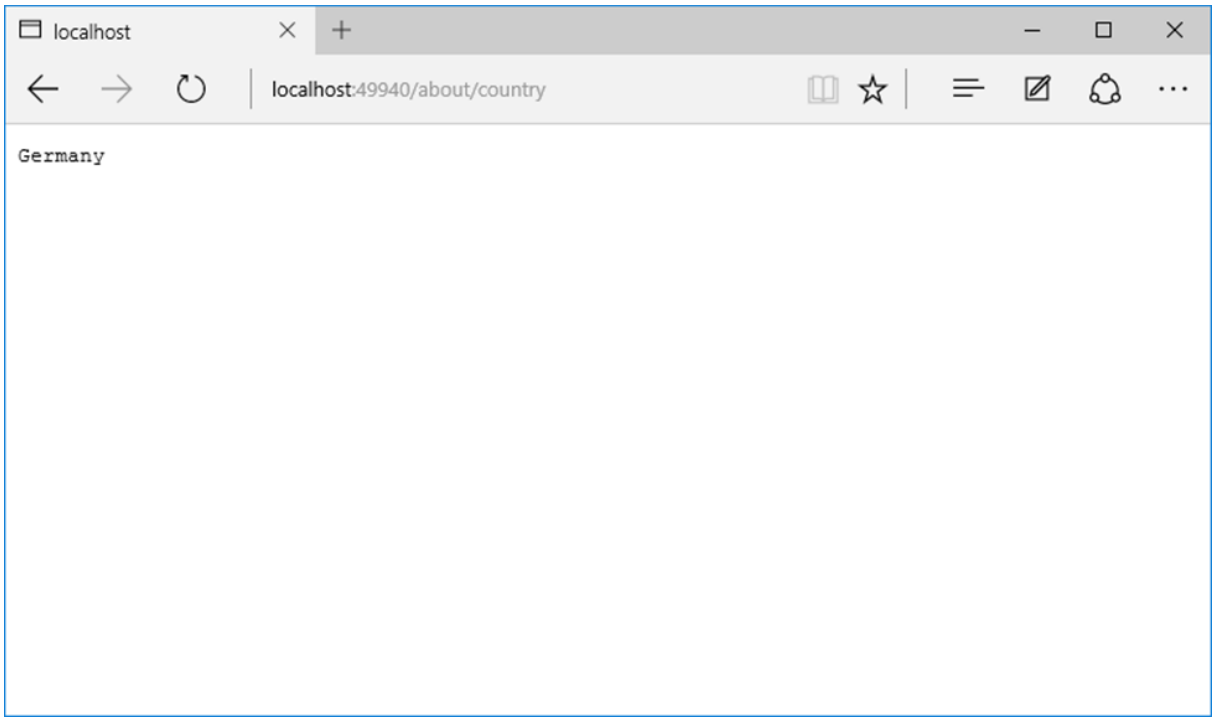
If you want a segment of the URL to contain the name of your controller, what you can do is instead of using the controller name explicitly, you can use a token controller inside the square brackets. This tells ASP.NET MVC to use the name of this controller in this position as shown in the following program.

```
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Controllers
{
    [Route("[controller]")]
    public class AboutController
    {
        [Route ("")]
        public string Phone()
        {
            return "+49-333-3333333";
        }

        [Route("[action]")]
        public string Country()
        {
            return "Germany";
        }
    }
}
```

This way, if you ever rename the controller, you don't have to remember to change the route. The same goes for an action and implicitly there is a slash (/) between the controller and the action. It is a hierarchical relationship between the controller and the action just like it is inside the URL. Let us save this controller again. Most probably, you will see the same results.



Let us specify the **/about/country**. This will allow you to get to that Country action.



14. ASP.NET Core — Action Results

In this chapter, we will discuss the Action Results. In the previous chapters, we have been using plain simple C# classes as controllers. These classes don't derive from a base class, and you can use this approach with MVC, but it is more common to derive a controller from a controller base class provided in the Microsoft.AspNet.Mvc namespace.

- This base class gives us access to lots of contextual information about a request, as well as methods that help us build results to send back to the client.
- You can send back simple strings and integers in a response. You can also send back complex objects like an object to represent a student or university or restaurant etc. and all the data associated with that object.
- These results are typically encapsulated into an object that implements the IActionResult interface.
- There are many different result types that implement this interface — result types that can contain models or the contents of a file for download.
- These different result types can allow us to send back JSON to a client or XML or a view that builds HTML.

Actions basically return different types of Action Results. The **ActionResult** class is the base for all the action results. The following is a list of different kind of action results and their behavior.

Name	Behavior
ContentResult	Returns a string.
FileContentResult	Returns file content.
FilePathResult	Returns file content.
FileStreamResult	Returns file content.
EmptyResult	Returns nothing.
JavaScriptResult	Returns script for execution.
JsonResult	Returns JSON formatted data
RedirectToResult	Redirects to the specified URL.
HttpUnauthorizedResult	Returns 403 HTTP Status code.

RedirectToRouteResult	Redirect to different action/ different controller action.
ViewResult	Received as a response for view engine.
PartialViewResult	Received as a response for view engine.

Example 1

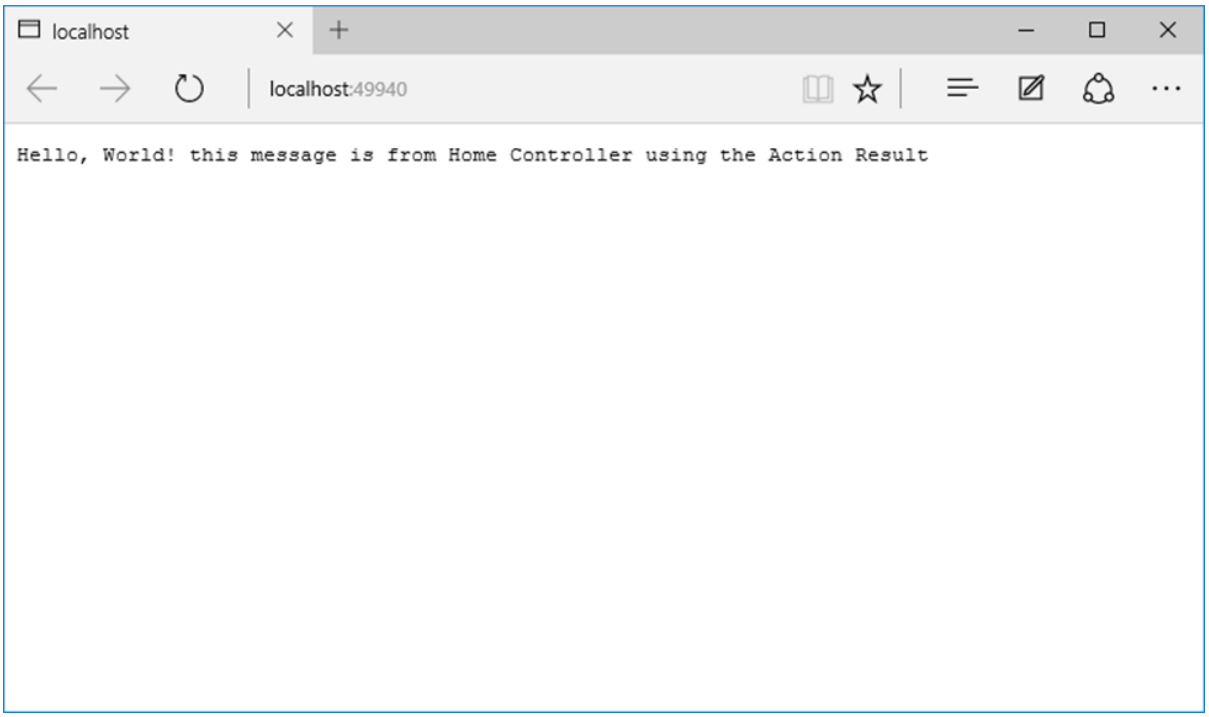
Let us perform a simple example by opening the **HomeController** class and derive it from the controller based class. This base class is in the **Microsoft.AspNet.Mvc** namespace. The following is the implementation of the HomeController class.

```
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
{
    public class HomeController : Controller
    {
        public ContentResult Index()
        {
            return Content("Hello, World! this message is from Home Controller
using the Action Result");
        }
    }
}
```

You can now see that the index method is returning the **ContentResult** which is one of the result types and all these result types implement ultimately an interface, which is the **ActionResult**.

In the Index method, we have passed a string into the Content method. This Content method produces a **ContentResult**; this means the Index method will now return ContentResult.

Let us save the **HomeController** class and run the application in the browser. It will produce the following page.

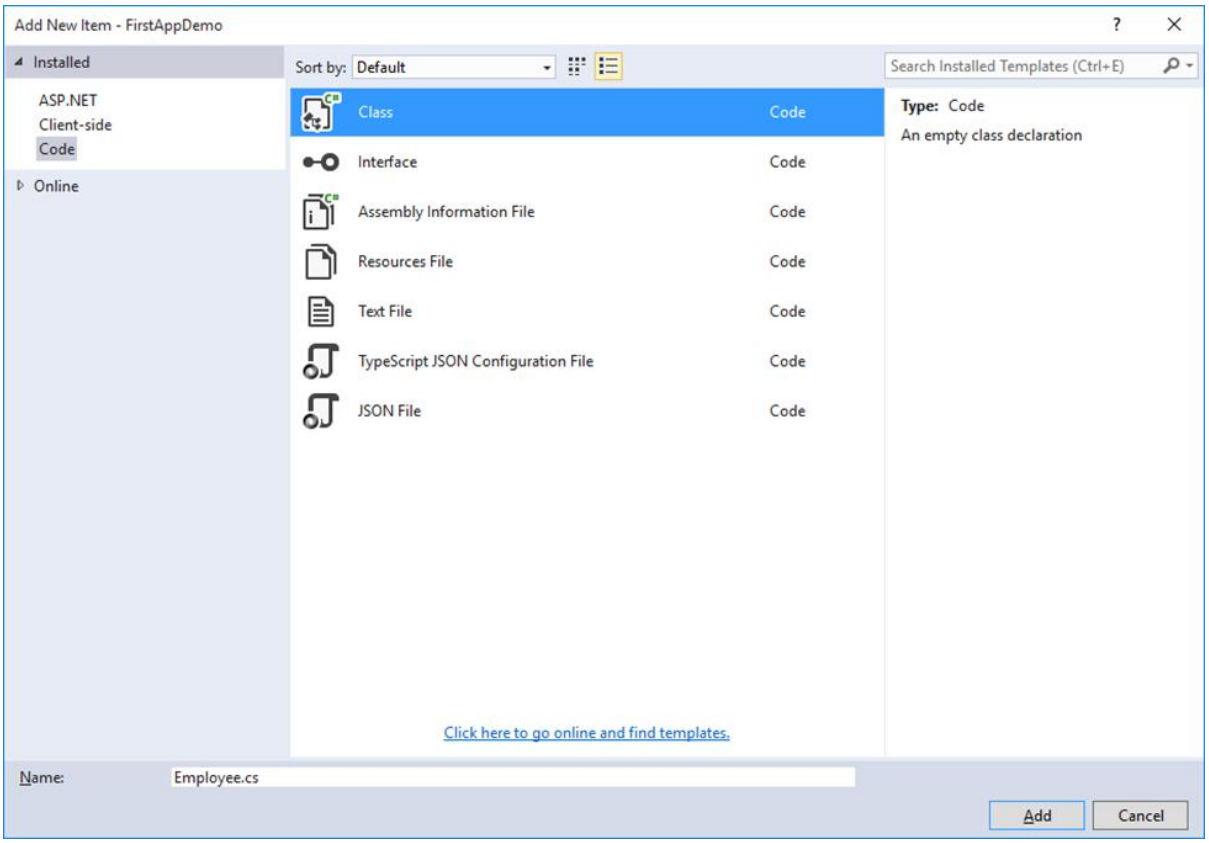


You can now see a response which doesn't look any different from the response we had before. It is still just going to be a plain text response.

- You might be wondering what is the advantage of using something that produces an **ActionResult**.
- The typical advantage is that it is just a formal way to encapsulate the decision of the controller.
- The controller decides what to do next, either return a string or HTML or return a model object that might be serialized into JSON etc.
- All that the controller needs to do is make that decision and the controller does not have to write directly into the response the results of its decision.
- It just needs to return the decision and then it is the framework that will take a result and understand how to transform that result into something that can be sent back over HTTP.

Example 2

Let us take another example. Create a new folder in the project and call it **Models**. Inside the Models folder, we want to add a class that can represent an Employee.



Enter **Employee.cs** in the Name field as in the above screenshot. Here, the implementation of the Employee class contains two properties.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Models
{
    public class Employee
    {
        public int ID { get; set; }
        public string Name { get; set}
    }
}
```

Inside the Index action method of **HomeController**, we want to return an Employee object. The following is the implementation of HomeController.

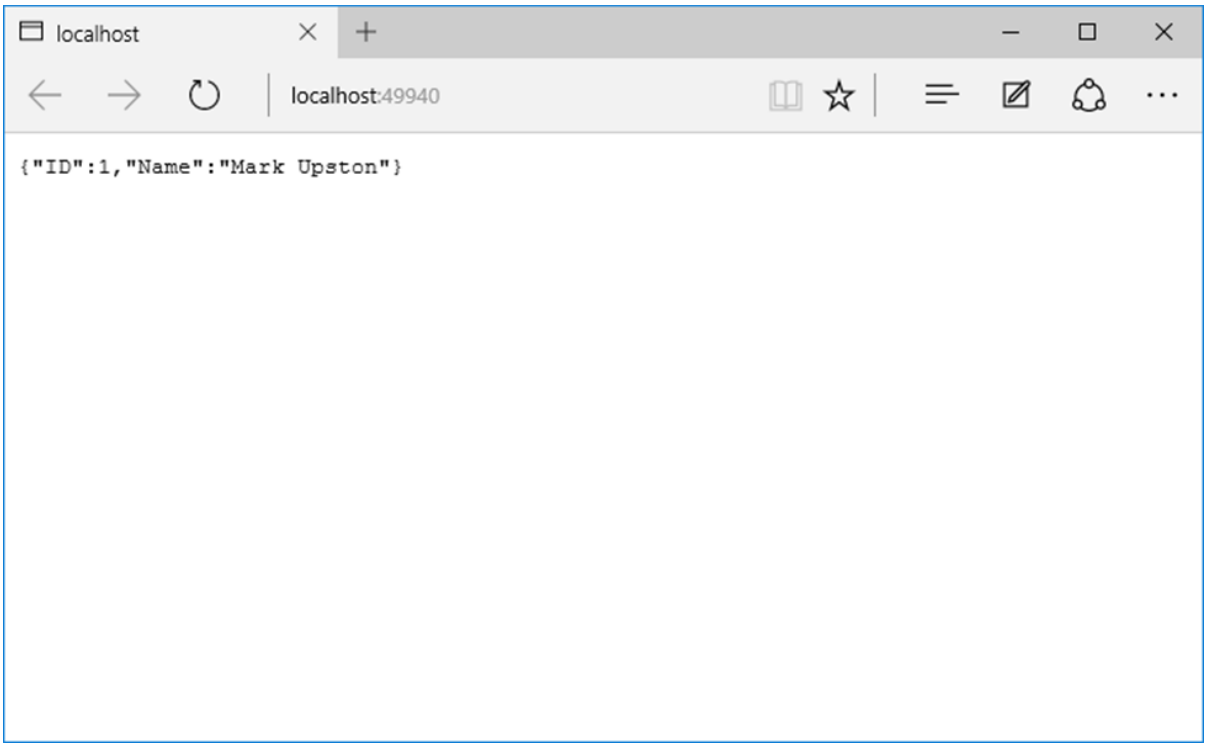
```
using FirstAppDemo.Models;
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
{
    public class HomeController : Controller
    {
        public ObjectResult Index()
        {
            var employee = new Employee { ID = 1, Name = "Mark Upston"};
            return new ObjectResult(employee);
        }
    }
}
```

Now, instead of returning the Content, we will return a different type of result which is known as **ObjectResult**. If we want an ObjectResult, we need to create or instantiate an ObjectResult and pass into it some **model** object.

- An ObjectResult is special in the MVC framework because when we return an ObjectResult, the MVC framework looks at this object. This object needs to be represented in the HTTP response.

- This object should be serialized into XML or JSON or some other format and ultimately, the decision will be made based on the configuration information that you give to the MVC at startup. If you don't configure anything, you just get some defaults, and the default is a JSON response.

Save all your files and refresh the browser. You will see the following output.



15. ASP.NET Core — Views

In an ASP.NET Core MVC application, there is nothing like a page and it also doesn't include anything that directly corresponds to a page when you specify a path in the URL. The closest thing to a page in an ASP.NET Core MVC application is known as a view.

- As you know that in ASP.NET MVC application, all incoming browser requests are handled by the controller and these requests are mapped to the controller actions.
- A controller action might return a view or it might also perform some other type of action such as redirecting to another controller action.
- With the MVC framework, the most popular method for creating HTML is to use the Razor view engine of ASP.NET MVC.
- To use this view engine, a controller action produces a **ViewResult** object, and a ViewResult can carry the name of the Razor view that we want to use.

- The view will be a file on the file system and the ViewResult can also carry along a model object to the view and the view can use this model object when it creates the HTML.
- When the MVC framework sees that your controller action produces a ViewResult, the framework will find the view on the file system, execute the view, which produces HTML, and it is this HTML which the framework sends back to the client.

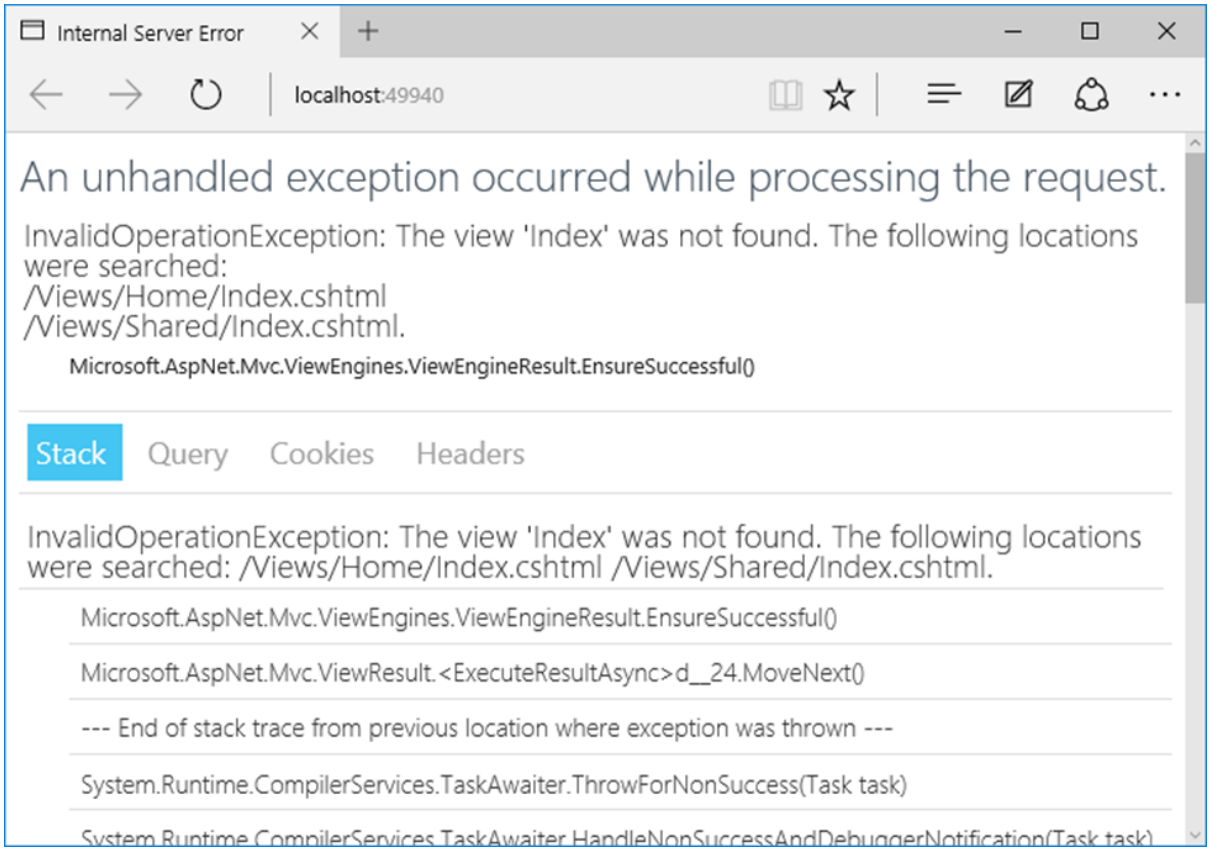
Example

Let us now take a simple example to understand how this works in our application by changing the HomeController Index method implementation as shown in the following program.

```
using FirstAppDemo.Models;
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
```

```
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            var employee = new Employee { ID = 1, Name = "Mark Upston"};
            return View();
        }
    }
}
```

Inside **HomeController**, instead of producing an **ObjectResult**, let us just return what the **View()** method returns. The View method doesn't return an ObjectResult. It creates a new ViewResult, so we will also change the return type of the Index method to ViewResult. The View method does accept some parameters here. We will invoke this method without any other parameter. Let us save your file and refresh your browser.



This is because the MVC framework has to go out and find that view but there is no view right now.

- Views by default in a C# ASP.NET project are files that have a *.cshtml extension and the views follow a specific convention. By default, all views live in a Views folder in the project.

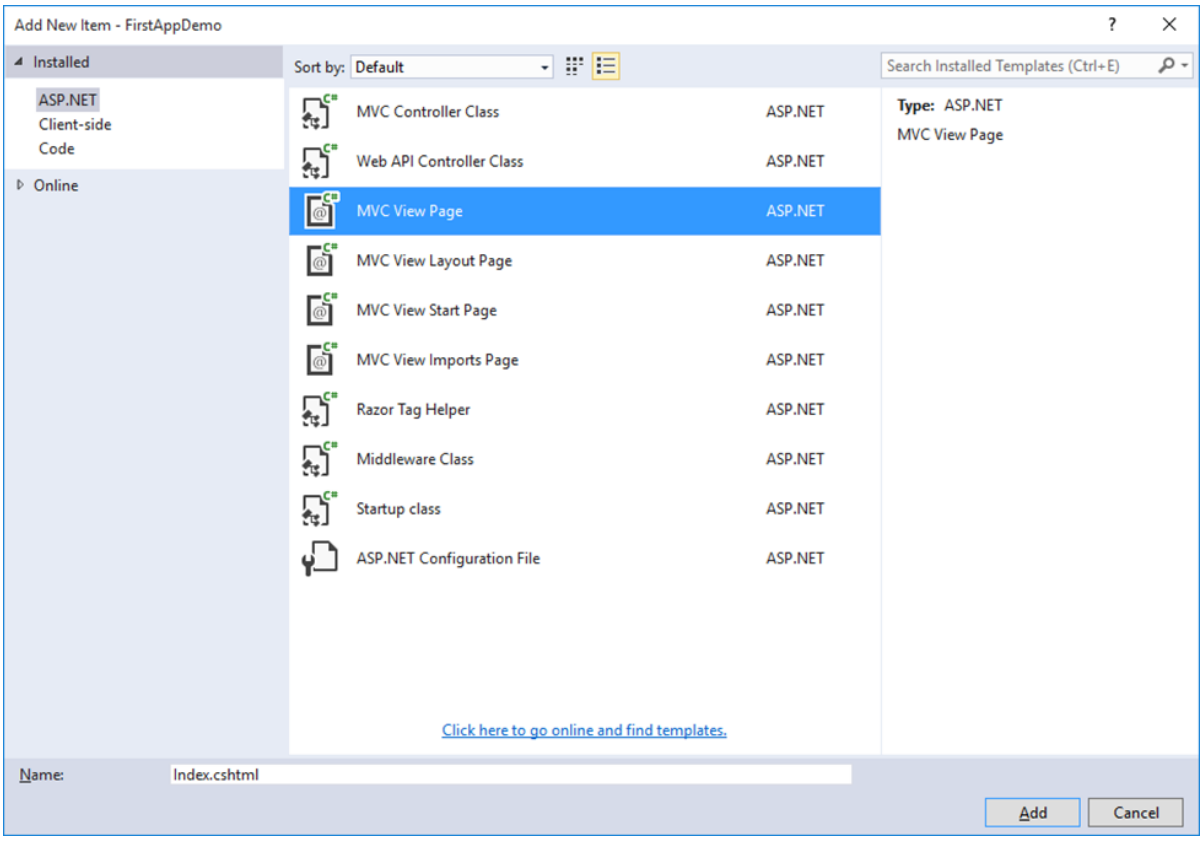
- The view location and the view file name will be derived by ASP.NET MVC if you don't give it any additional information.

- If we need to render a view from the Index action of the HomeController, the first place that the MVC framework will look for that view is inside the Views folder.

- It will go into a Home folder and then look for a file called Index.cshtml — the file name starts with Index because we are in the Index action.

- The MVC framework will also look in a Shared folder and views that you place inside the Shared folder, you can use them anywhere in the application.

In order for our view result to work properly, let us create this Index.cshtml file in the correct location. So in our project, we first need to add a folder that will contain all of our views and call it Views. Inside the Views folder, we will add another folder for views that are associated with our HomeController and call that folder Home. Right-click on the Home folder and select **Add > New Item.**



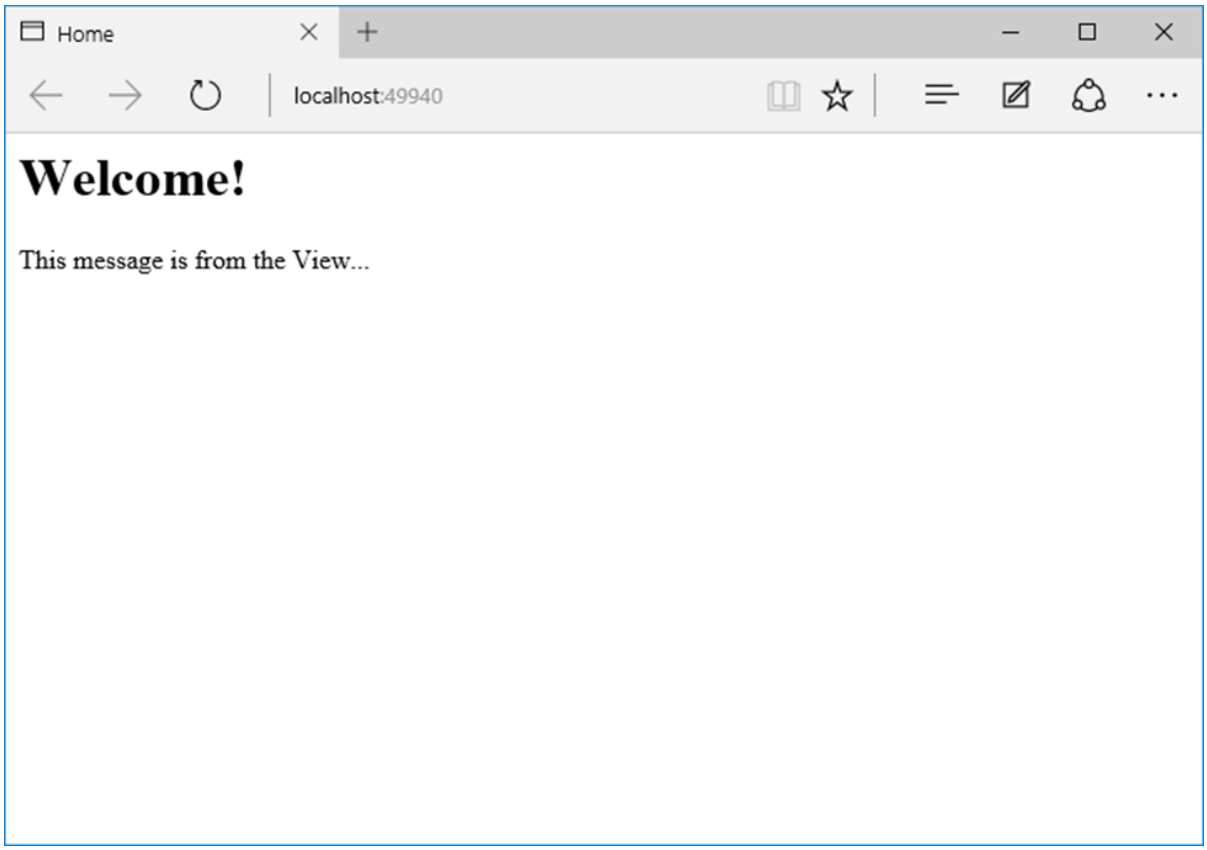
In the left pane, select the MVC View Page and enter **index.cshtml** in the name field and click on the Add button.

Let us add the following code in the index.cshtml file.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Home</title>
</head>
```

```
<body>
    <h1>Welcome!</h1>
    <div>
        This message is from the View...
    </div>

</body>
</html>
```

You can now see a ***.cshtml file**. It can contain HTML markup and any markup that we have in this file will be sent directly to the client. Save this file and refresh your browser.



Now the Home controller via a ViewResult has rendered this view to the client and all of the markup that is in that index.cshtml file, that is what was sent to the client.

Let us go back to the HomeController and the View method. This View method has a couple of different overloads, and pass the employee model as parameter.

```
using FirstAppDemo.Models;
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
```

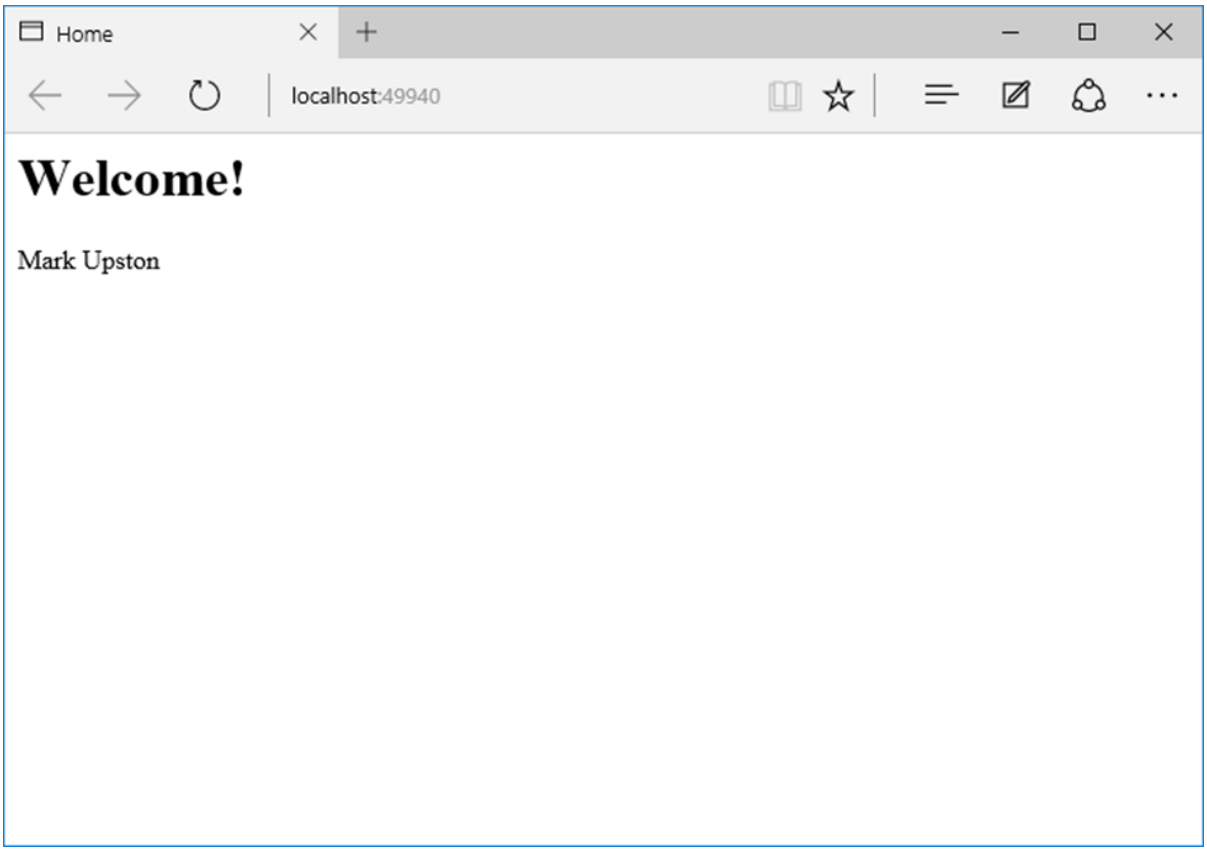
```
using System.Threading.Tasks;

namespace FirstAppdemo.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            var employee = new Employee { ID = 1, Name = "Mark Upston"};
            return View(employee);
        }
    }
}
```

The View method that just takes a model object and that will use the default view, which is Index. Here we just want to pass in that model information and use that model inside Index.cshtml as shown in the following program.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Home</title>
</head>
<body>
    <h1>Welcome!</h1>
    <div>
        @Model.Name
    </div>

</body>
</html>
```

When we use the @ sign in a **Razor view**, then the Razor view engine is going to treat whatever you type as a C# expression. Razor view contains some built-in members that we can access inside the C# expressions. One of the most important members is the Model. When you say @Model, then you will get the model object that you have passed into the view from the controller. So here the @Model.Name will display the employee name inside the view.

Let us now save all the files. After this, refresh your browser to see the following output.



You can now see the employee name as in the above screenshot.

16. ASP.NET Core — Setup Entity Framework

In this chapter, we will set up and configure our application to save and read data from a SQL Server database.

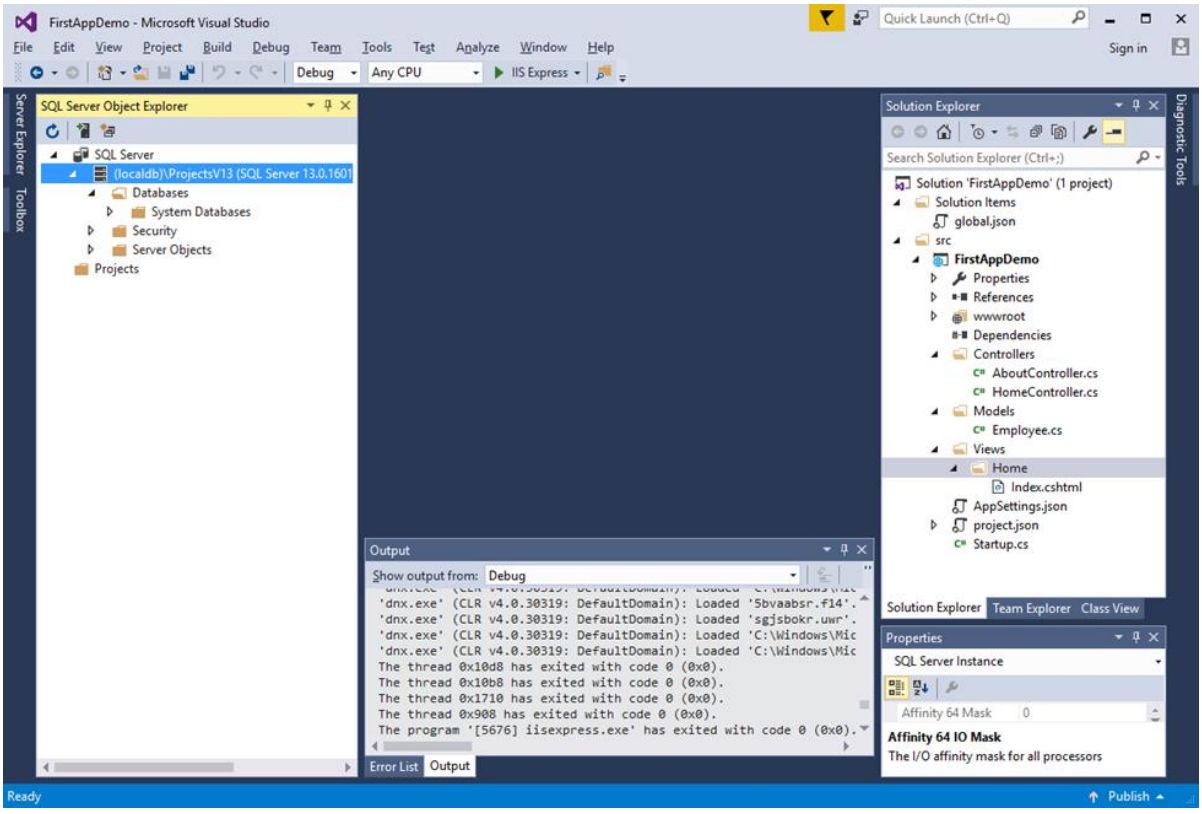
To work with a database, we are going to use the Entity Framework, which is freshly rewritten to work with the new .NET Framework. If you have worked with EF in the past, you will see many familiar pieces.

- In this application, we will use the SQL Server LocalDB. If you are not comfortable with the SQL Server, you can use any database that you like such as local database, remote database, as long as you have permission to create a new database on the instance.

- LocalDB is a special edition of SQL Server that is optimized for developers.

- Visual Studio 2015 and even its Community edition will install LocalDB by default.

To check the LocalDB, go to the **View > SQL Server Object Explorer** menu option in Visual Studio.

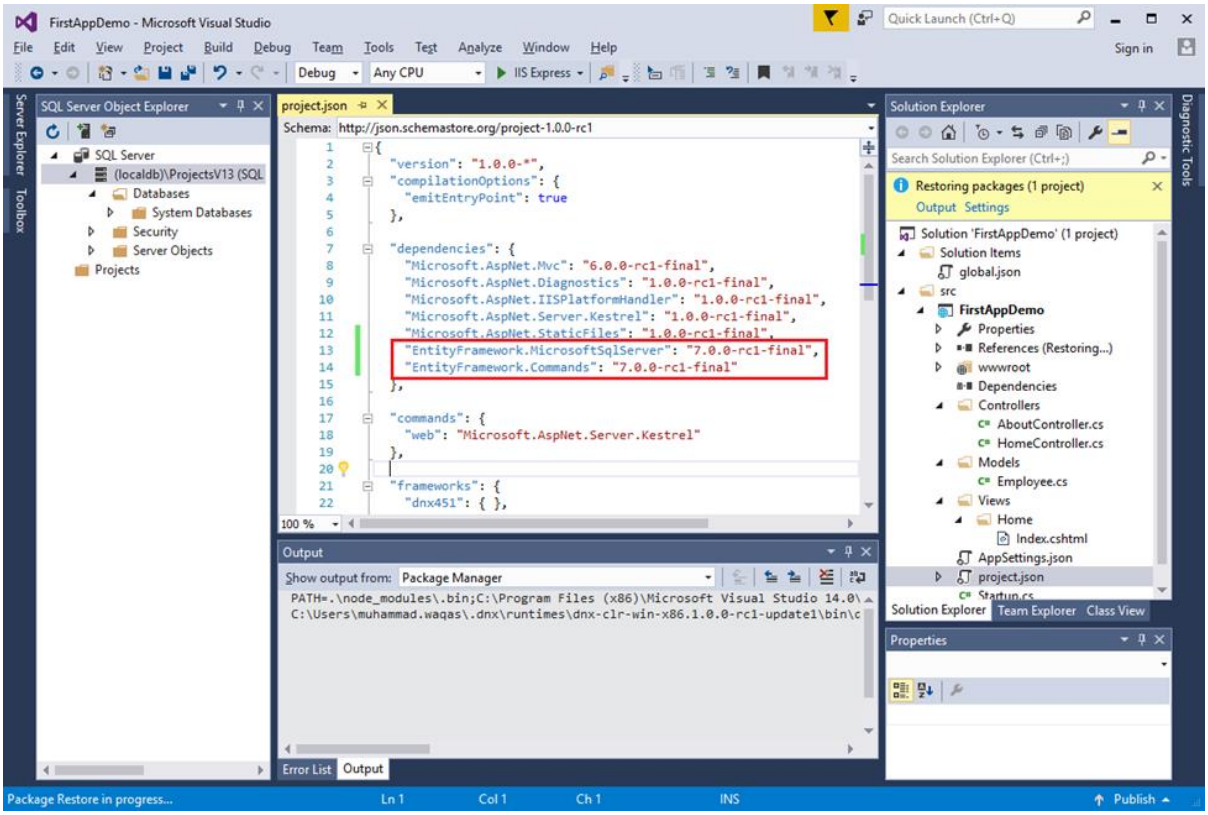


This is a great tool if you have to work with SQL Server because it allows you to explore databases and browse data and even create data inside a database. When you first open it, it might take a little time, but it should connect to the LocalDB automatically.

Install Entity Framework

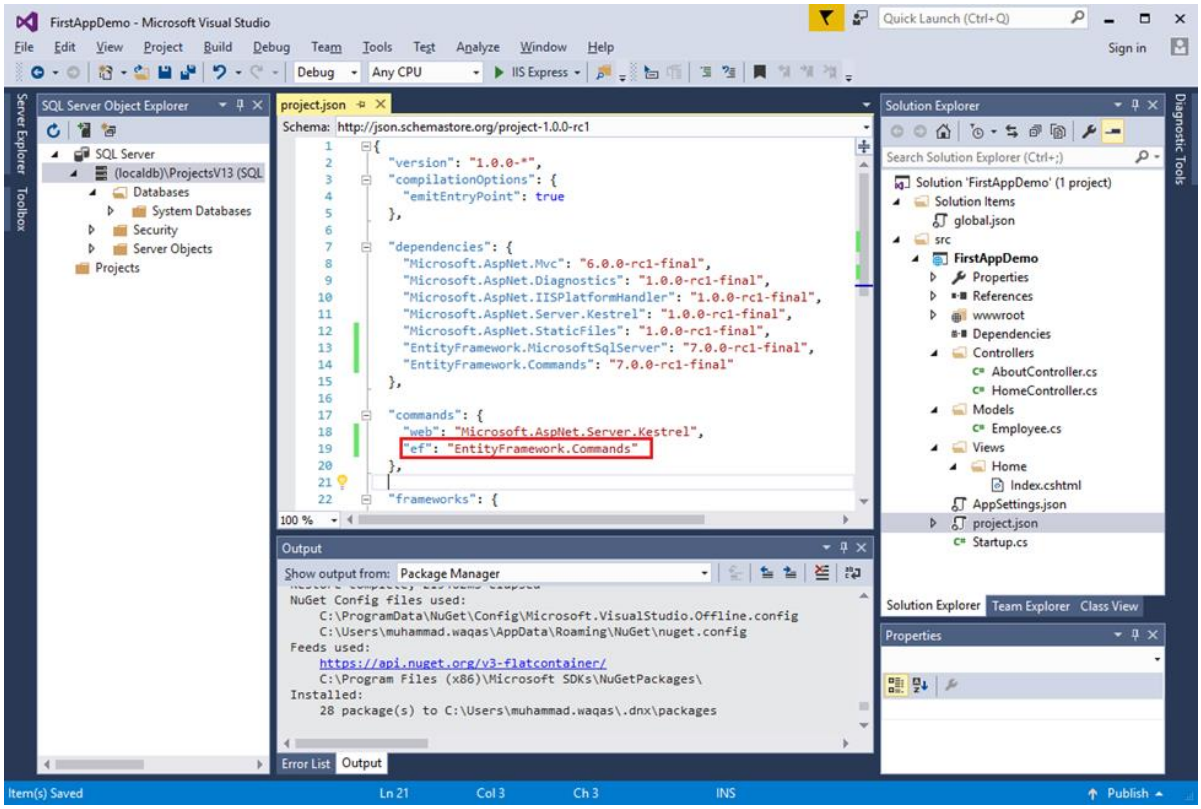
The first step in using the Entity Framework is to install the Entity Framework NuGet package from the NuGet package manager or by editing the **project.json** file directly.

Let us now edit the project.json file directly by adding the following two packages.



The **EntityFramework.Commands** package helps us perform tasks with the Entity Framework like creating a database schema based on our C# Entity classes and these tasks are available from a command line tool where the logic lives inside the EntityFramework.Commands package.

In order to use this command line tool, we need to make an additional entry into the commands section of project.json as shown in the following screenshot.



We have just called it "ef" and that will map to this EntityFramework.Commands package. We can use this "ef" to get access to some of the logic that is available inside EntityFramework.Commands.

The following is the implementation of the project.json file.

```
{
  "version": "1.0.0-*",
  "compilationOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Diagnostics": "1.0.0-rc1-final",
    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "EntityFramework.Commands": "7.0.0-rc1-final"
  },
```

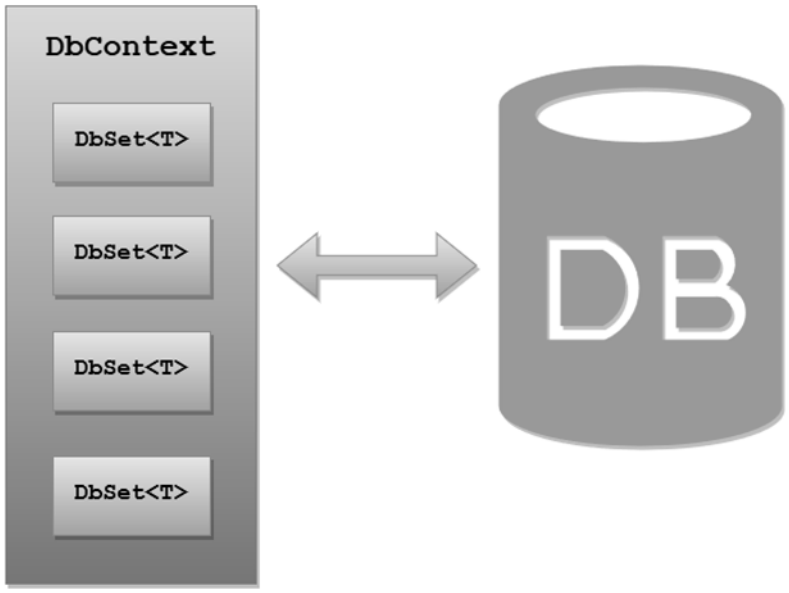
```
  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
  "publishExclude": [
    "**.user",
    "**.vspscc"
  ]
}
```

17. ASP.NET Core — DbContext

The Entity Framework enables you to query, insert, update, and delete data, using Common Language Runtime (CLR) objects known as entities. The Entity Framework maps the entities and relationships that are defined in your model to a database. It also provides facilities to –

- Materialize data returned from the database as entity objects.
- Track changes that were made to the objects.
- Handle concurrency.
- Propagate object changes back to the database.
- Bind objects to controls.

The primary class that is responsible for interacting with data as objects is the DbContext. The recommended way to work with context is to define a class that derives from the DbContext and exposes the DbSet properties that represent collections of the specified entities in the context.

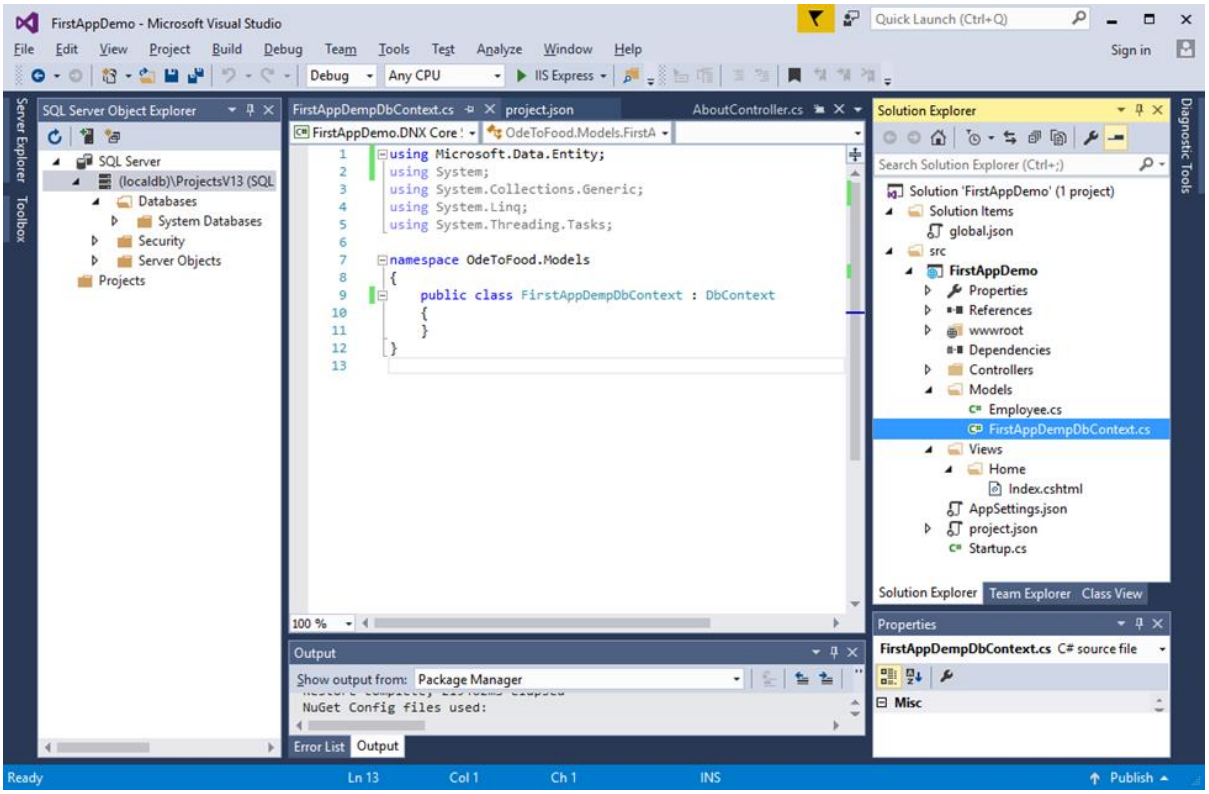


Logically, a DBContext maps to a specific database that has a schema that the DBContext understands. And on that DBContext class, you can create properties that are type DbSet<T>. The generic type parameter T will be a type of entity like Employee is an entity in the FirstAppDemo application.

Example

Let us take a simple example, wherein we will create a DbContext class. Here, we need to add a new class in Models folder and call it **FirstAppDempDbContext**. Even though this

class is not a model in itself, it does put together all our Models so that we can use them with the database.



Inherit your context class from the DbContext class which is in Miscrosoft.Data.Entity namespace. Now implement a DbSet of Employee on that class.

Each DbSet will map to a table in the database. If you have a property DbSet of employee, and the name of that property is Employees, the Entity Framework will by default look for an Employees table inside your database.

```
using FirstAppDemo.Models;
using Microsoft.Data.Entity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace OdeToFood.Models
{
    public class FirstAppDemoDbContext : DbContext
    {
        public DbSet<Employee> Employees { get; set; }
    }
}
```

The implementation is very simple because we only have a single model to work with. We need only one property, the **DbSet** of Employee and we can name this property **Employees**. This is the only code we need for this class.

Let us now insert this class directly into controllers, and the controllers could then use **FirstAppDemoDbContext** to query the database. We will simplify all these by adding a new class to the HomeController class in which we implement methods to Add employee and Get employee as shown in the following program.

```
using Microsoft.AspNet.Mvc;
using FirstAppDemo.ViewModels;
using FirstAppDemo.Services;
using FirstAppDemo.Entities;
using FirstAppDemo.Models;
using System.Collections.Generic;
using System.Linq;

namespace FirstAppDemo.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            var model = new HomePageViewModel();
            using (var context = new FirstAppDemoDbContext())
            {
                SQLEmployeeData sqlData = new SQLEmployeeData(context);
                model.Employees = sqlData.GetAll();
            }

            return View(model);
        }
    }

    public class SQLEmployeeData
    {
        private FirstAppDemoDbContext _context { get; set; }
        public SQLEmployeeData(FirstAppDemoDbContext context)
        {
            _context = context;
```

```
        }
        public void Add(Employee emp)
        {
            _context.Add(emp);
            _context.SaveChanges();
        }
        public Employee Get(int ID)
        {
            return _context.Employees.FirstOrDefault(e => e.Id == ID);
        }
        public IEnumerable<Employee> GetAll()
        {
            return _context.Employees.ToList<Employee>();
        }
    }
    public class HomePageViewModel
    {
        public IEnumerable<Employee> Employees { get; set; }
    }
}
```

In the above SQLEmployeeData class, you can see that we have defined the Add method which will add a new employee object to the context and then it will save the changes. In the Get method, it will return an employee based on the ID. Whereas, in the GetAll method it will return the list of all the employees in the database.

Configuring the Entity Framework Services

To have a usable Entity Framework DBContext, we need to change the configuration of the application. We will need to add a connection string so that our DBContext knows which server to go to and which database to query.

- We will put the connection string in a JSON configuration file.
- We also need to add some more services during the ConfigureServices method of the Startup class.
- The Entity Framework, just like ASP.NET and the MVC framework, the Entity Framework relies on dependency injection, and for injection to work, the runtime needs to know about the various services that the Entity Framework uses.
- There is an easy configuration API that will add all the default services that we need.

Let us go to the AppSettings.json file and add the connections string as shown in the following program.

```
{
  "message": "Hello, World! this message is from configuration file...",
  "database": {
    "connection": "Data Source=(localdb)\\mssqllocaldb;Initial
Catalog=FirstAppDemo"
  }
}
```

Let us now go to the Startup class where we need to add some additional services for the Entity Framework to work properly. Specifically, there are three things that we need to do that are related to the Entity Framework:

- We need to add the core Entity Framework services.
- We also need to add the SQL Server-related Entity Framework services.
- We need to tell the Entity Framework about our DBContext.

All this can be done through methods that are available as extension methods on **IServiceCollection** as shown in the following program.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<FirstAppDemoDbContext>(option =>
option.UseSqlServer(Configuration["database:connection"]));
}
```

- The first method is the **AddEntityFramework**. This will add the core Entity Framework services, the default services.
- But since the Entity Framework is now designed to work with different sorts of databases, including non-relational databases, we need to make a second call to tell the Entity Framework to add its default SQL Server-related services.
- Then we also need to tell the Entity Framework about my **DBContext** class so it can construct instances of that class appropriately and we can do that through a third method, the **AddDbContext** method.
- This one takes a generic type parameter where we specify the type of the DBContext derived class, the **FirstAppDemoDbContext**.

- Inside the AddDbContext, we need to describe the options for our DBContext.
- This can be done by a **lambda expression**; it is an action where we receive an option parameter and the Entity Framework can support the different databases. All we need to do is, tell the Entity Framework that this particular DBContext is going to **UseSqlServer**.
- This method requires a parameter which is the **connectionString** to use.

The following is the complete implementation of the **Startup.cs** file.

```
using Microsoft.AspNet.Mvc;
using FirstAppDemo.ViewModels;
using FirstAppDemo.Services;
using FirstAppDemo.Entities;
using FirstAppDemo.Models;
using System.Collections.Generic;
using System.Linq;

namespace FirstAppDemo.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            var employee = new Employee { Id = 1, Name = "Mark Upston1" };
            using (var context = new FirstAppDemoDbContext())
            {
                SQLEmployeeData sqlData = new SQLEmployeeData(context);
                sqlData.Add(employee);
            }
            //var employee = new Employee { ID = 1, Name = "Mark Upston" };
            return View(employee);
        }
    }

    public class SQLEmployeeData
    {
        private FirstAppDemoDbContext _context { get; set; }

        public SQLEmployeeData(FirstAppDemoDbContext context)
        {
```

```
            _context = context;
        }

        public void Add(Employee emp)
        {
            _context.Add(emp);
            _context.SaveChanges();
        }
        public Employee Get(int ID)
        {
            return _context.Employees.FirstOrDefault(e => e.Id == ID);
        }
        public IEnumerable<Employee> GetAll()
        {
            return _context.Employees.ToList<Employee>();
        }
    }
}
```

Now we need to set up the database. One way to get a database set up is to use the Entity Framework to create the database and this is a two-step process:

The First Step

This involves the following:

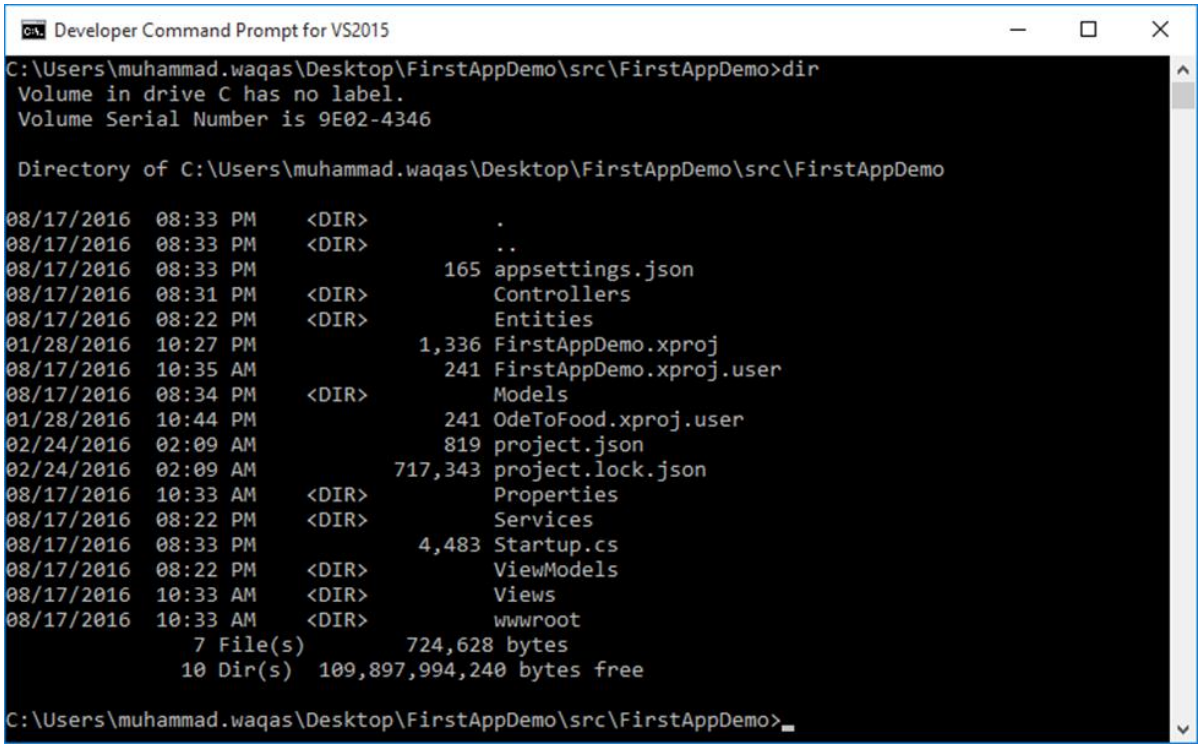
- Adding migration code to our project.
- The migration code is **C#** code. This can be executed to create a database in a database schema.
- The Entity Framework can generate this migration code for us.
- The Entity Framework looks at the database and at our models and figures out what the schema changes are required to make the application work.
- So when we add additional models or make changes to the existing models, like the Employee class, we can continue to add migrations to our project and keep our database schema in sync.

The Second Step

This involves the following:

- Here, we need to explicitly apply those migrations to update a database.
- Both of these tasks can be achieved by using some easy commands from a console window.
- We have made project.json.
- That is why we have made project.json to add a command where "**ef**" maps to EntityFramework.Commands.

Let us open the Developer Command Prompt for Visual Studio to run the commands that we need to add the migrations and apply the migrations. The easiest way to do this is to go to the application root directory.

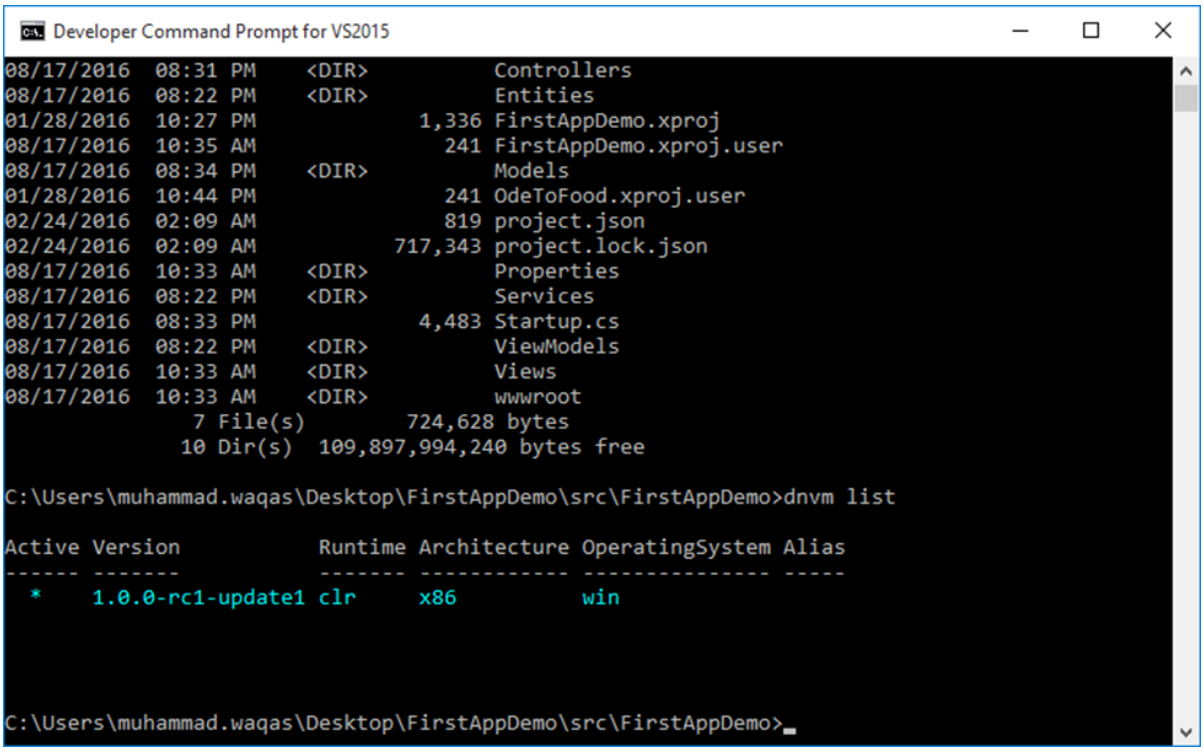


If you are in the folder that has the project.json file, then you are in the correct folder. Here, we need to execute a command known as dnvm. This is the .NET version manager which will tell the system what runtime we want to use.

Let us now use the following command.

```
dnvm list
```

You will see the following output when you press Enter.

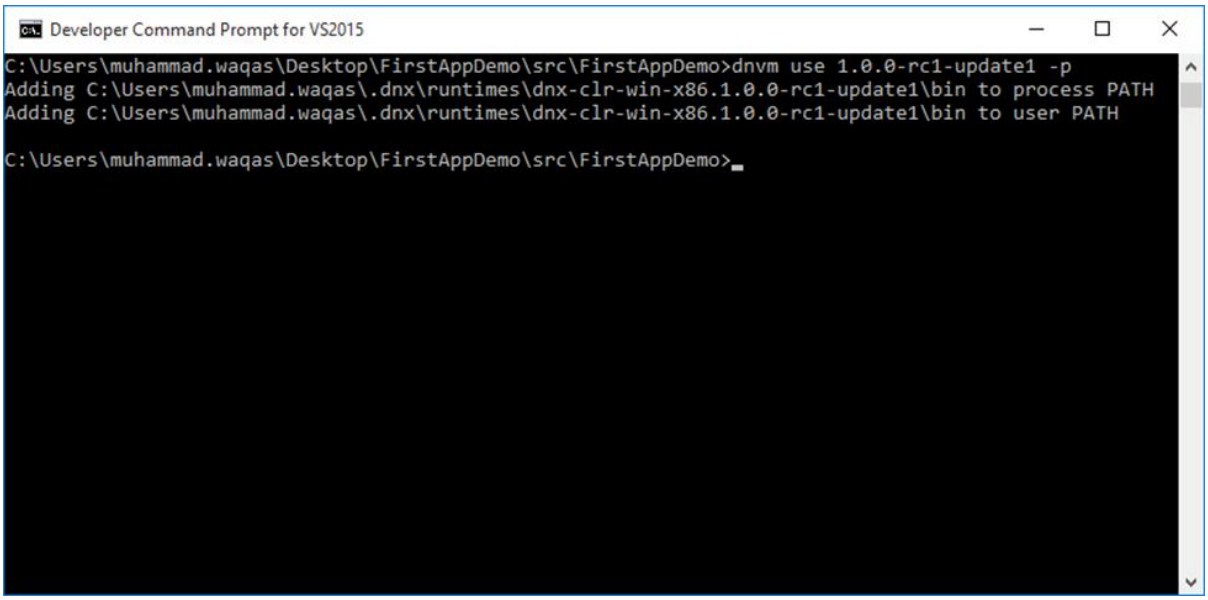


We need to tell the **dnvm** that we want to use the specific runtimes. This will give us access to the dotnet command or the dnx command that we want to execute.

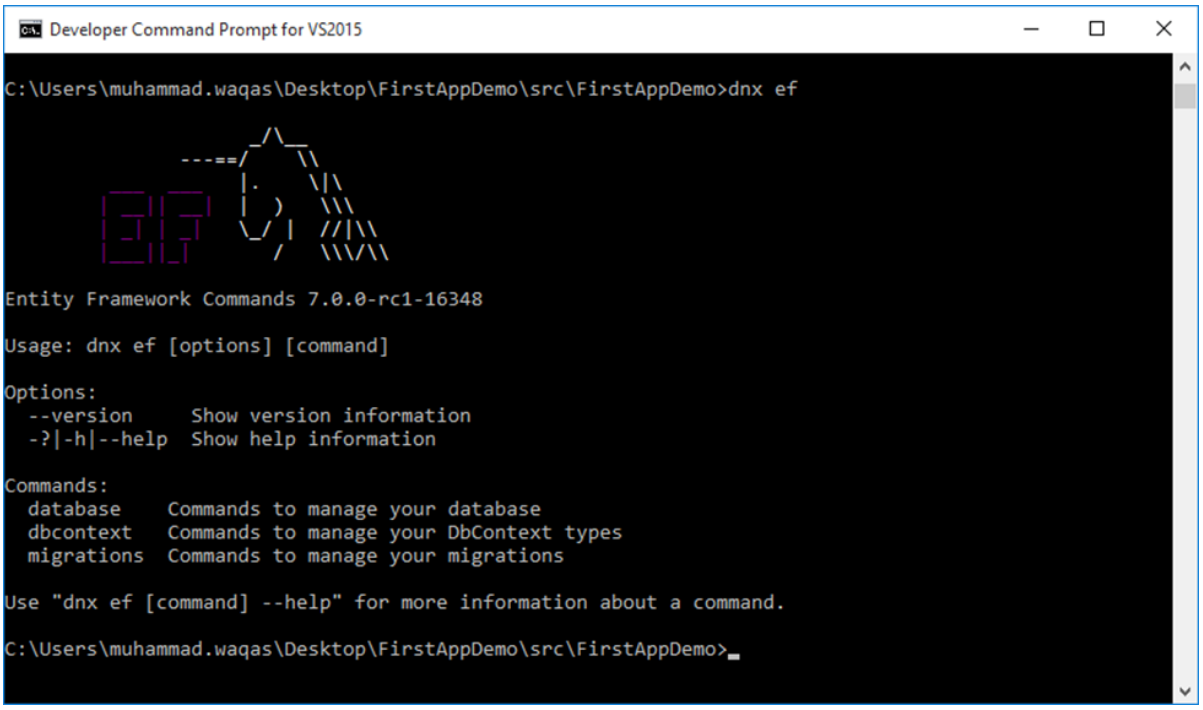
Execute the following command.

```
dnvm use1.0.0-rc1-update1 -p
```

Press Enter.



dnvm will set up our path and the environment variables to include a bin directory that will give us access to this dnx utility. Let us execute the **dnx ef** command.

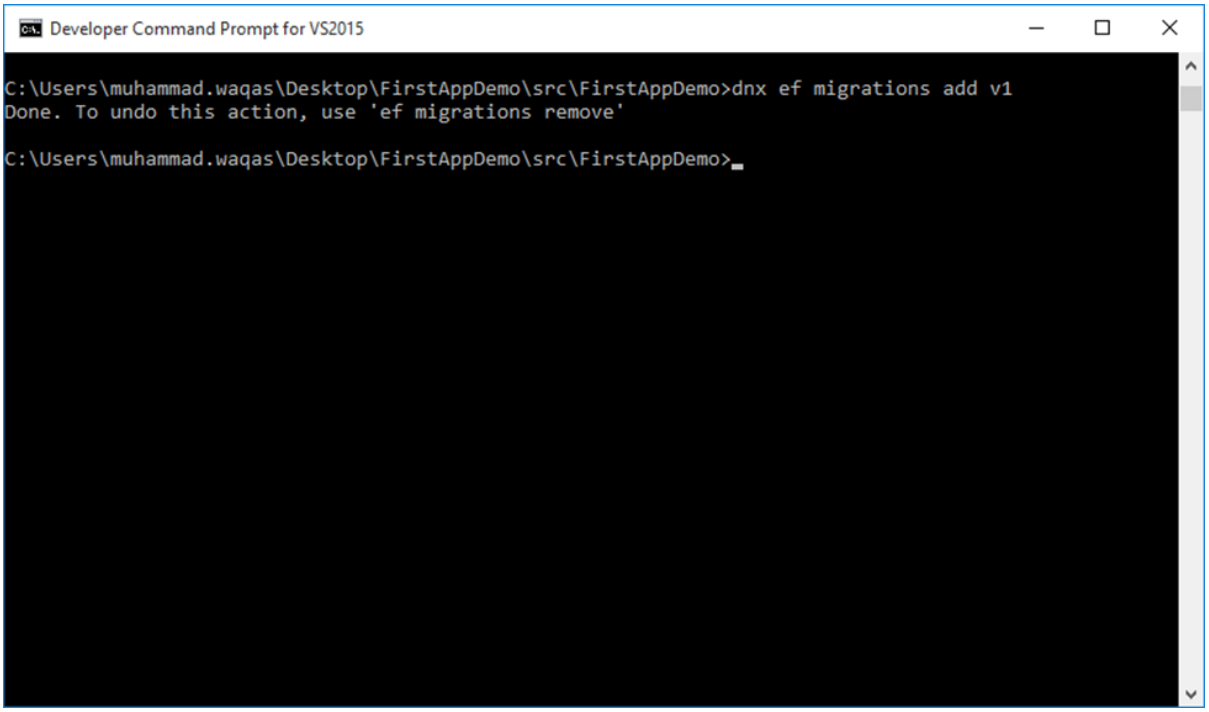


This is the .NET execution environment, using dnx, we can call the commands that we have listed in our project.json file. Executing these commands is generally very easy. When you type dnx ef, you will get a help screen. You don't have to remember all the options. You can see the available commands from the Entity Framework Commands and there are three of them.

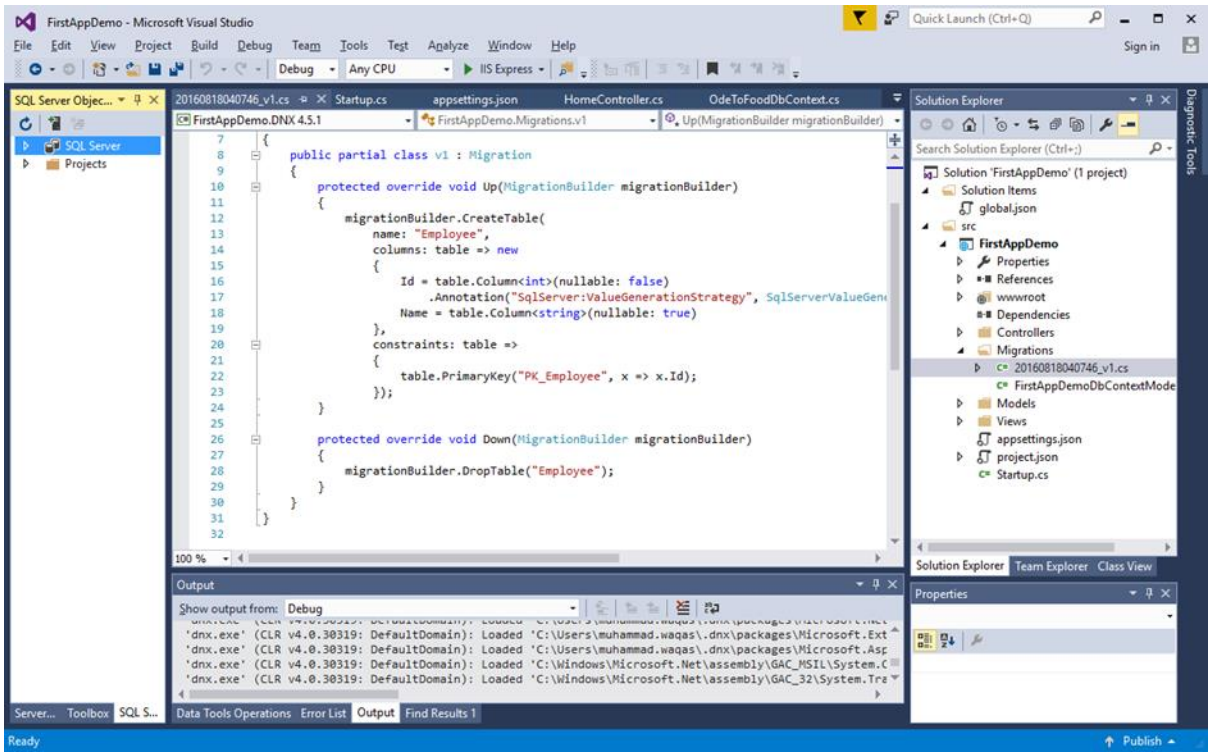
First, we need to add migration to execute the following command.

```
dnx ef migrations add v1
```

Press Enter.



The Entity Framework will find that context and look at the models that are inside. It will know that there is no previous migration and so it is going to generate the first migration. Here, v1 is the version 1 of the database. It will create a new folder in Solution Explorer and generate code.



A migration is essentially a C# code that is used to generate SQL commands to modify the schema in a SQL database.

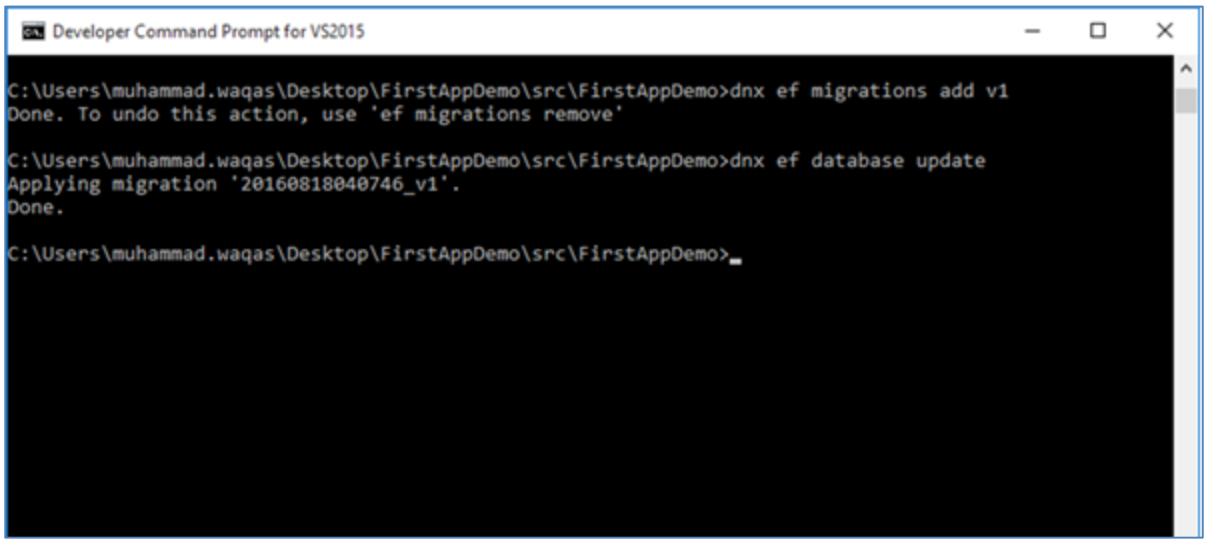
```
using System;
using System.Collections.Generic;
using Microsoft.Data.Entity.Migrations;
using Microsoft.Data.Entity.Metadata;

namespace FirstAppDemo.Migrations
{
    public partial class v1 : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Employee",
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
```

```
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    Name = table.Column<string>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Employee", x => x.Id);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable("Employee");
        }
    }
}
```

You can see it will create a table called Employees.

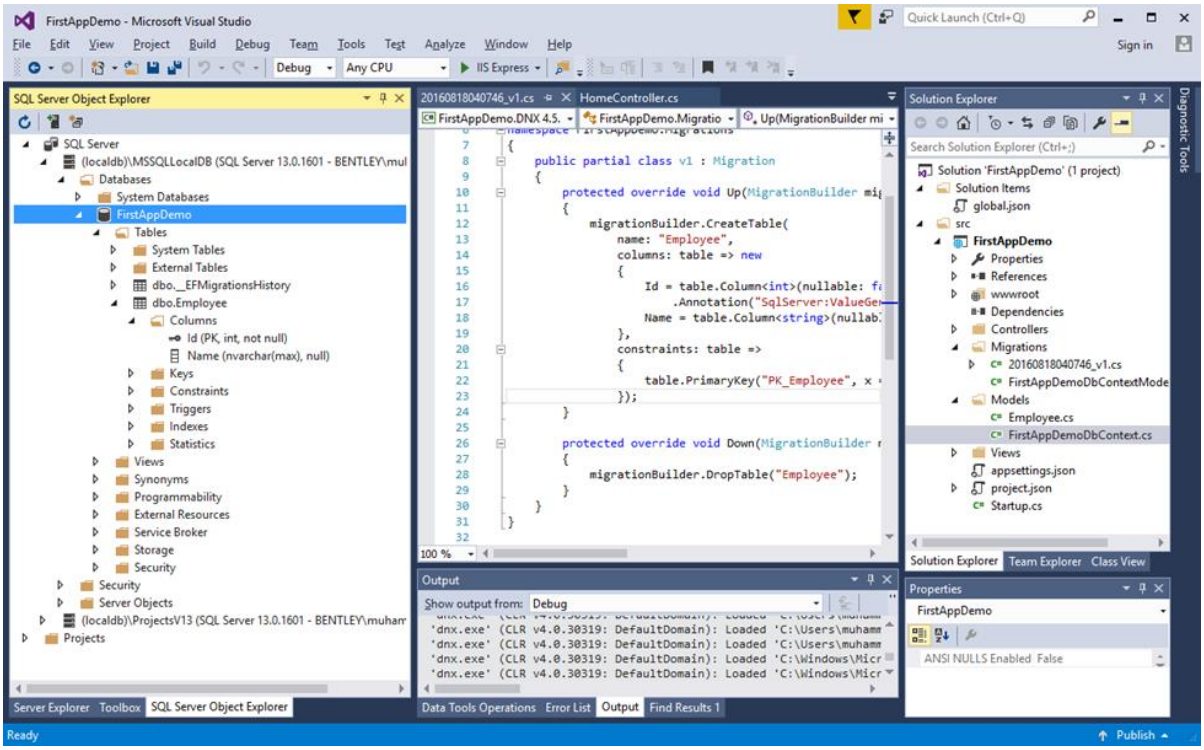
- This table should have two columns — an ID, and a Name column.
- By convention, when the Entity Framework sees that you have a property called Id, it will make that property or, rather, make that column a primary key in the database.
- Here, we will use the SQL Server. By default, the Entity Framework will make that an IdentityColumn, which means the SQL Server will generate the IDs for us.

Let us apply these IDs to a database by typing the "**dnx ef database update**" command.



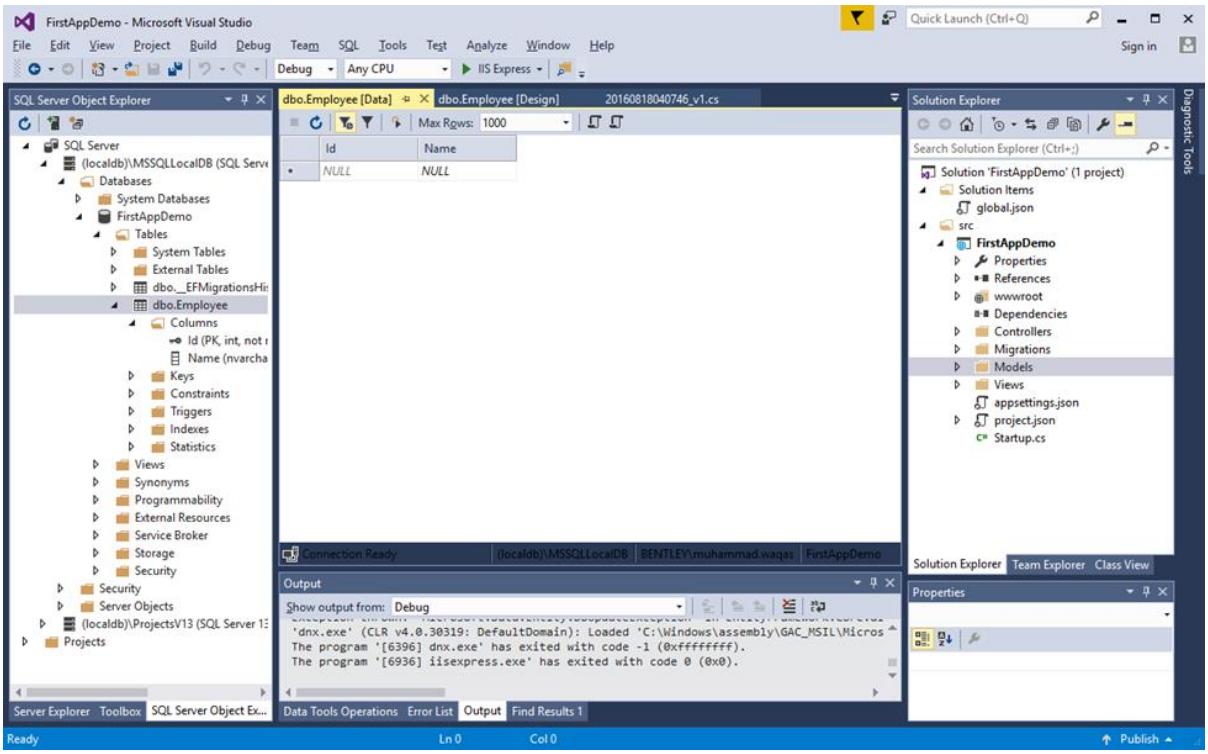
You can see that the command has applied migration.

Let us now go to the SQL Server Object Explorer, and refresh the databases, you can now see we have a FirstAppDemo database.



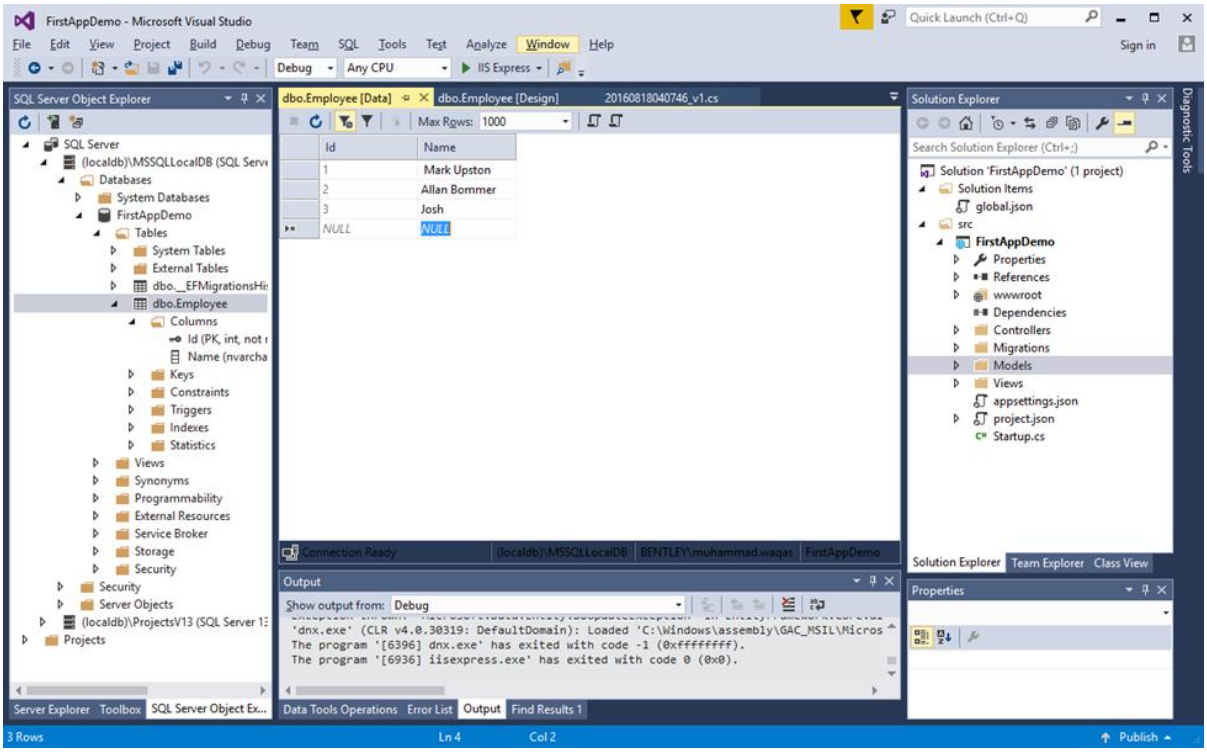
You can also see our Employee table and we can even look at the columns for that table in which the ID column is the primary key.

Let us right-click on the dbo.Employee table and select View Data.



Before we run the application, let us add some data. When we launch the application, we should see some data from the database.

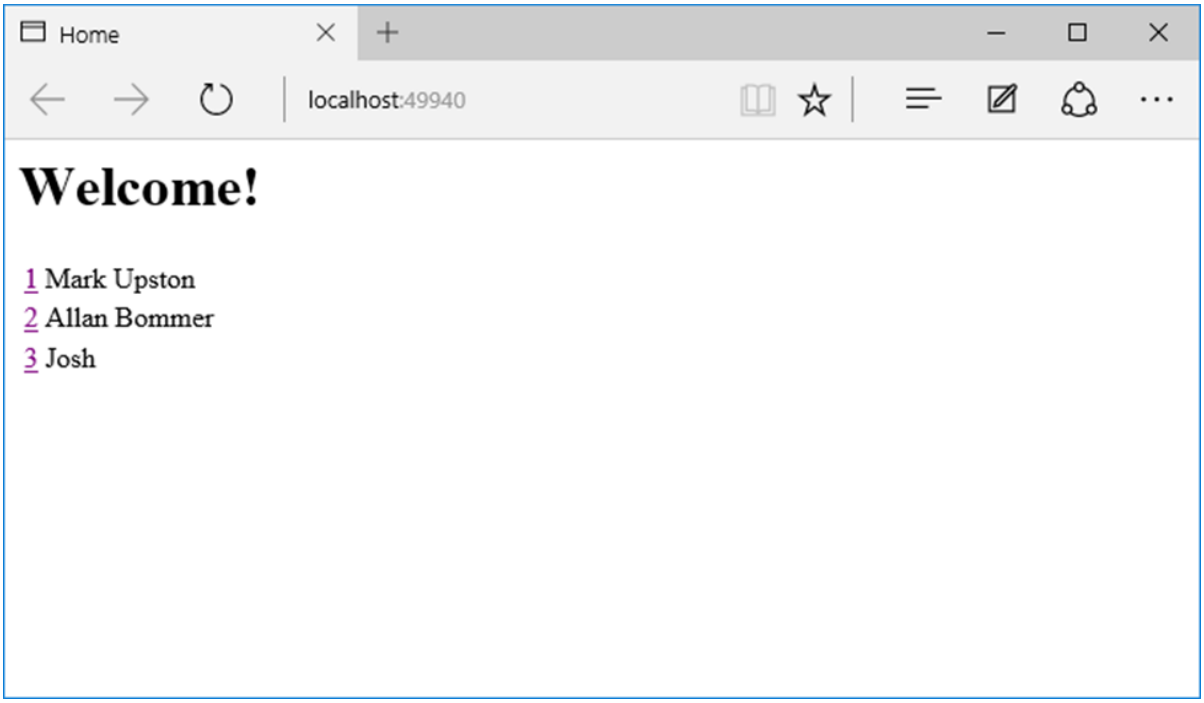
Let us just add a couple of rows of data here.



Let us now update the index.cshtml file. It shows all the data in tabular form.

```
@model FirstAppDemo.Controllers.HomePageViewModel
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Home</title>
</head>
<body>
    <h1>Welcome!</h1>
    <table>
        @foreach (var employee in Model.Employees)
        {
            <tr>
                <td>
                    @Html.ActionLink(employee.Id.ToString(), "Details", new {
id=employee.Id })
                </td>
                <td>@employee.Name</td>
            </tr>
        }
    </table>
</body>
</html>
```

Once you run the application, it should produce the following output.



18. ASP.NET Core — Razor Layout Views

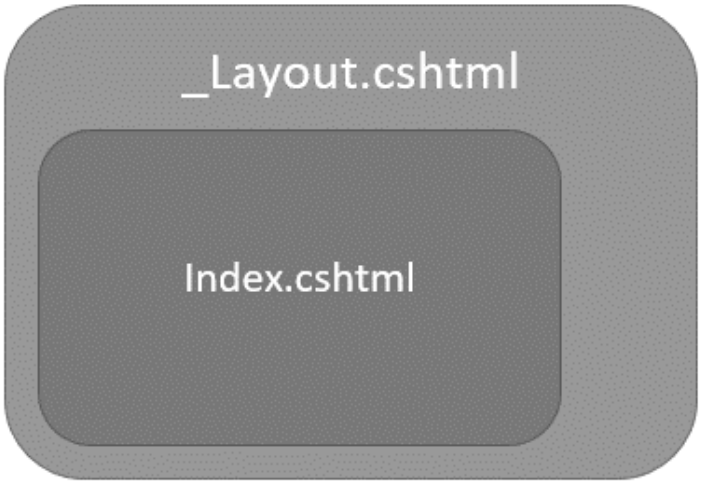
In this chapter, we will understand the Razor Layout Views. Most websites and web applications will want to create pages that present some common elements.

- You typically have a top area on every page where you display a logo and a navigational menu.
- You might also have a sidebar with additional links and information and probably a footer at the bottom of the page with some content.
- Every page of the application will want to have these common factors. Here, we make use of the Layout view to avoid duplication of factors in every page that we write.

Layout View

Let us now understand what a Layout View is.

- A Layout view is a Razor view with a ***.cshtml** extension. You have a choice to name a Layout view the way you want. In this chapter, we will be using a Layout view with the name **_Layout.cshtml**.
- This is a common name for a Layout view, and the leading underscore is not required. That is just a convention that many developers follow to identify a view that is not a view; you render this as a view result from a controller action.
- It is a special kind of view, but once we have a Layout view, we can set up our controller views like the Index view for the home page.



- We can set up this view to render inside the Layout view at a specific location.
- This Layout view approach means that the Index.cshtml doesn't need to know anything about the logo or the top level navigation.
- The Index view only needs to render the specific content for the model the controller action gives this view and the Layout view takes care of everything else.

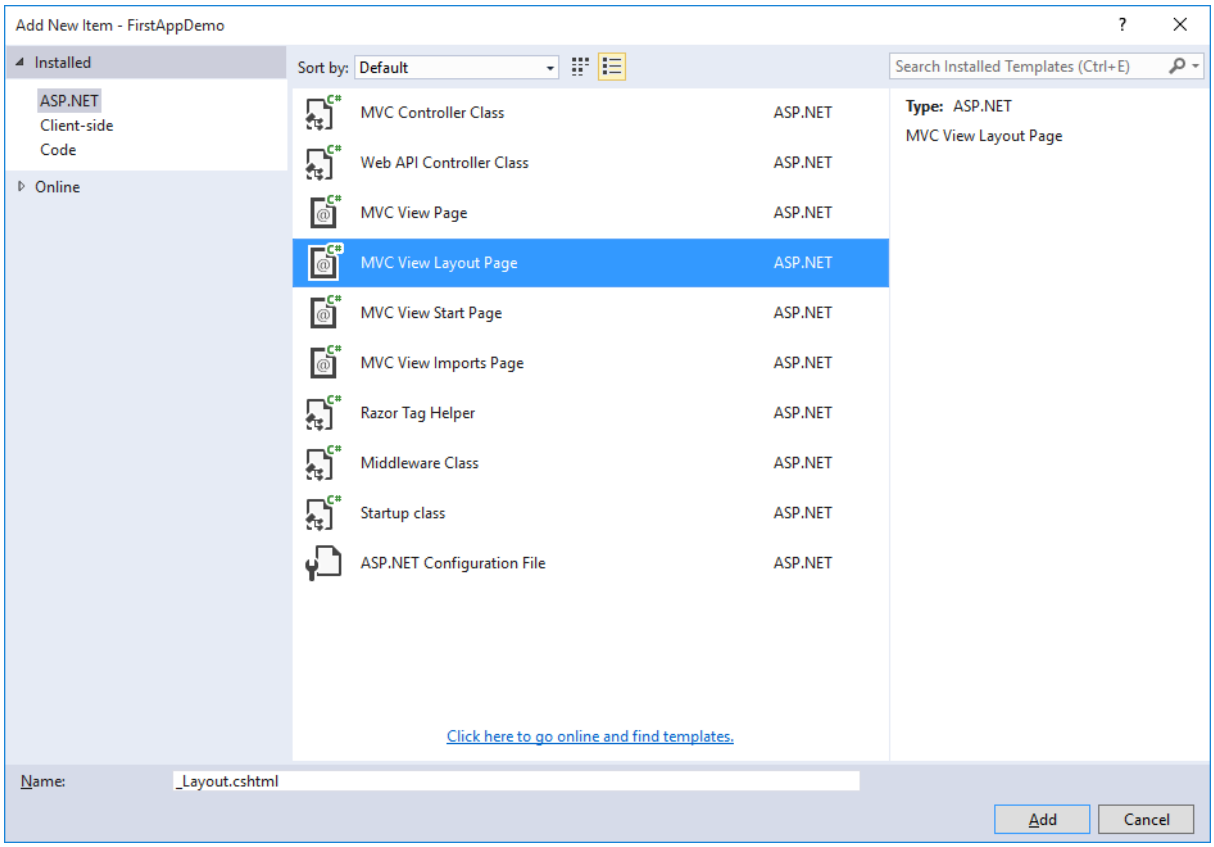
Example

Let us take a simple example.

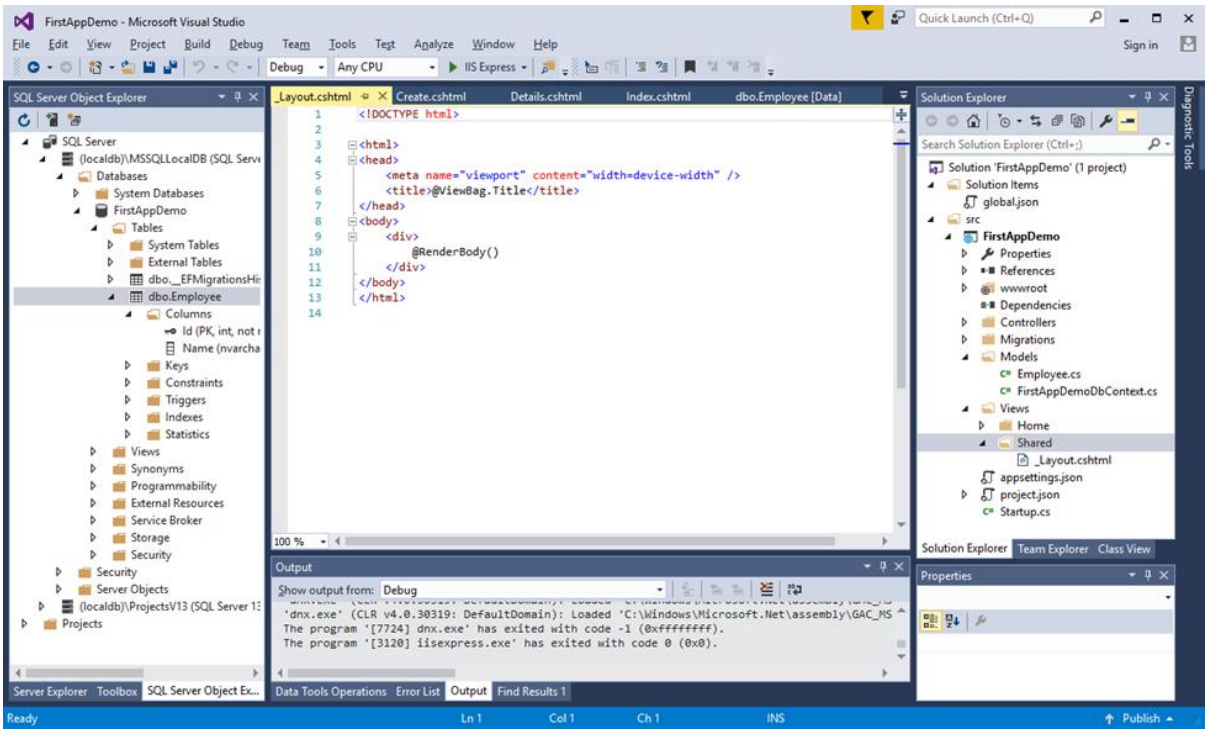
If you have multiple views, then you will see that all the views will contain some amount of duplicate markup. They all will have an opening **HTML tag**, a **head tag**, and a **body tag**.

Even though we don't have a navigational menu in this application, chances are in a real application it might be also available and we don't want to duplicate that markup in every view.

Let us create a Layout view and we will add the Layout view to a new folder named Shared inside the **Views** folder. This is a conventional folder that the MVC framework knows about. It knows that views inside here might be used by multiple controllers across the application. Let us right-click on the Shared folder and select Add > New Item…



In the middle pane, select the MVC View Layout Page. The default name here is _Layout.cshtml. Choose the Layout view that you want to use at runtime depending on the user. Now, click on the Add button. This is what you will get by default for your new Layout view.



We want the Layout view to be responsible for managing the head and the body. Now, as this view is in a Razor view we can use the C# expressions. We can still add literal text. We now have a div that displays DateTime.Now. Let us now just add Year.

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @DateTime.Now
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

In the above code, you will see expressions like **RenderBody** and **ViewBag.Title**. When an MVC controller action renders the Index view, and with it there is a layout page involved; then the Index view and the HTML it produces will be placed in the Index view.

This is where the method call to RenderBody exists. We can expect all of the content views throughout our application to appear inside the div where RenderBody is called.

The other expression inside this file is the ViewBag.Title. ViewBag is a data structure that can be added to any property and any data that you want to into the ViewBag. We can add a ViewBag.Title, ViewBag.CurrentDate or any properties that we want on the ViewBag.

Let us now go to the index.cshtml file.

```
@model FirstAppDemo.Controllers.HomePageViewModel
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Home</title>
</head>
<body>
    <h1>Welcome!</h1>
    <table>
        @foreach (var employee in Model.Employees)
        {
            <tr>
                <td>
                    @Html.ActionLink(employee.Id.ToString(), "Details", new {
id=employee.Id })
                </td>
                <td>@employee.Name</td>
            </tr>
        }
    </table>
</body>
</html>
```

Remove the markup that we no longer need inside the Index view. So, we can remove things like the HTML tag and the head tag. We also don't need the opening body element or the closing tags as shown in the following program.

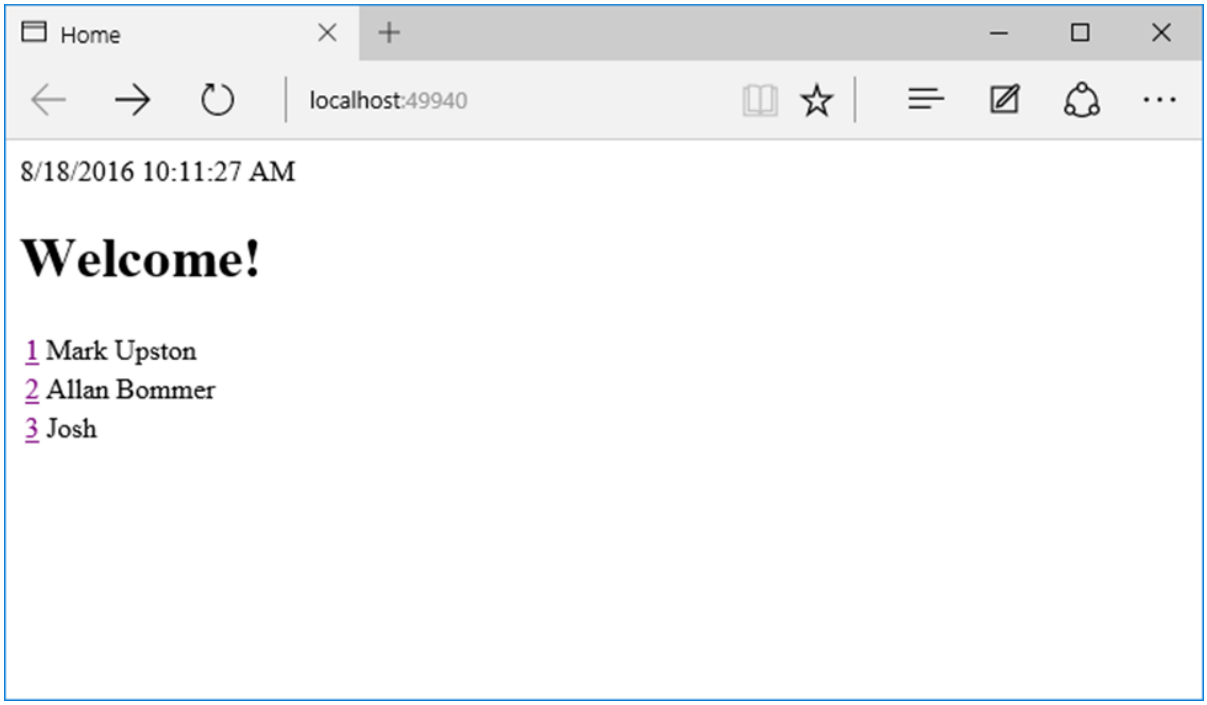
```
@model FirstAppDemo.Controllers.HomePageViewModel

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

```
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>
                @Html.ActionLink(employee.Id.ToString(), "Details", new { id =
employee.Id })
            </td>
            <td>@employee.Name</td>
        </tr>
    }
</table>
```

We still need to do two things:

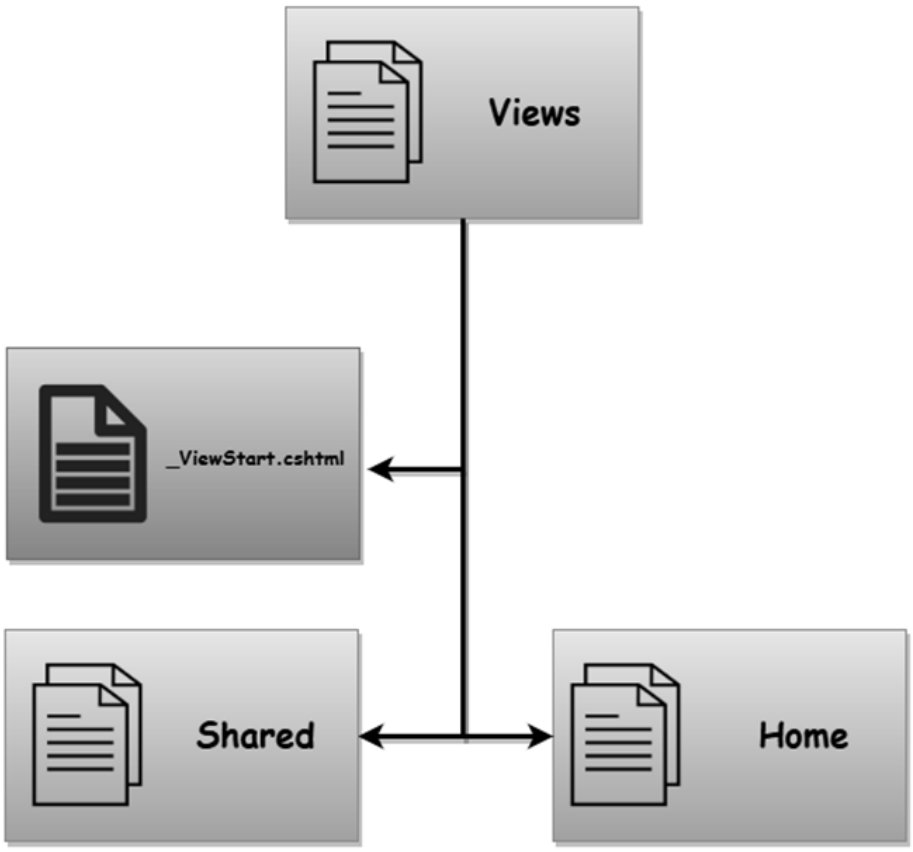
- First, we need to tell the MVC framework that we want to use the Layout view from this view.
- Second, we need to set the appropriate title by adding some information into the ViewBag as shown in the above code.

Let us save all the files and run the application. Once you run the application, you will see the following home page.



19. ASP.NET Core — Razor View Start

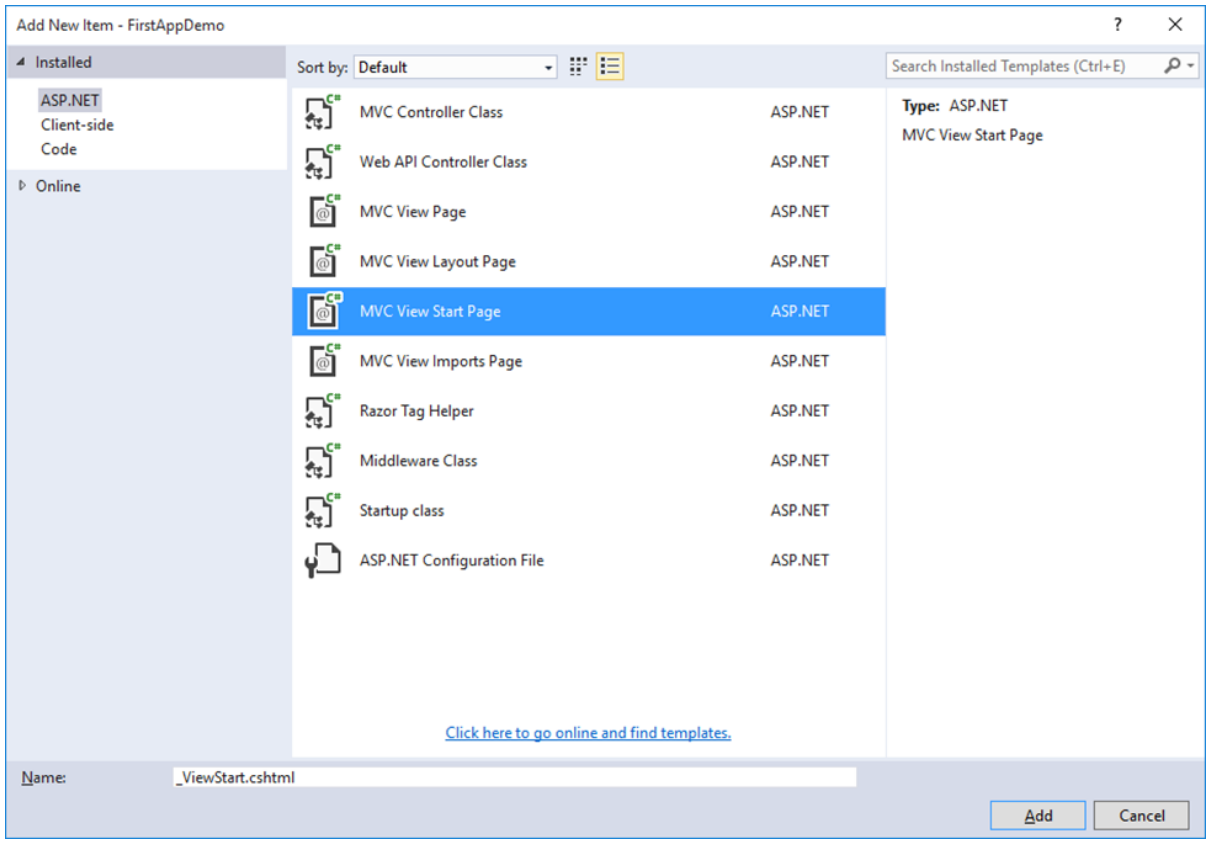
In this chapter, we will discuss the Razor View Start. The Razor view engine in MVC has a convention where it will look for any file with the name **_ViewStart.cshtml** and execute the code inside this file. before executing the code inside an individual view.



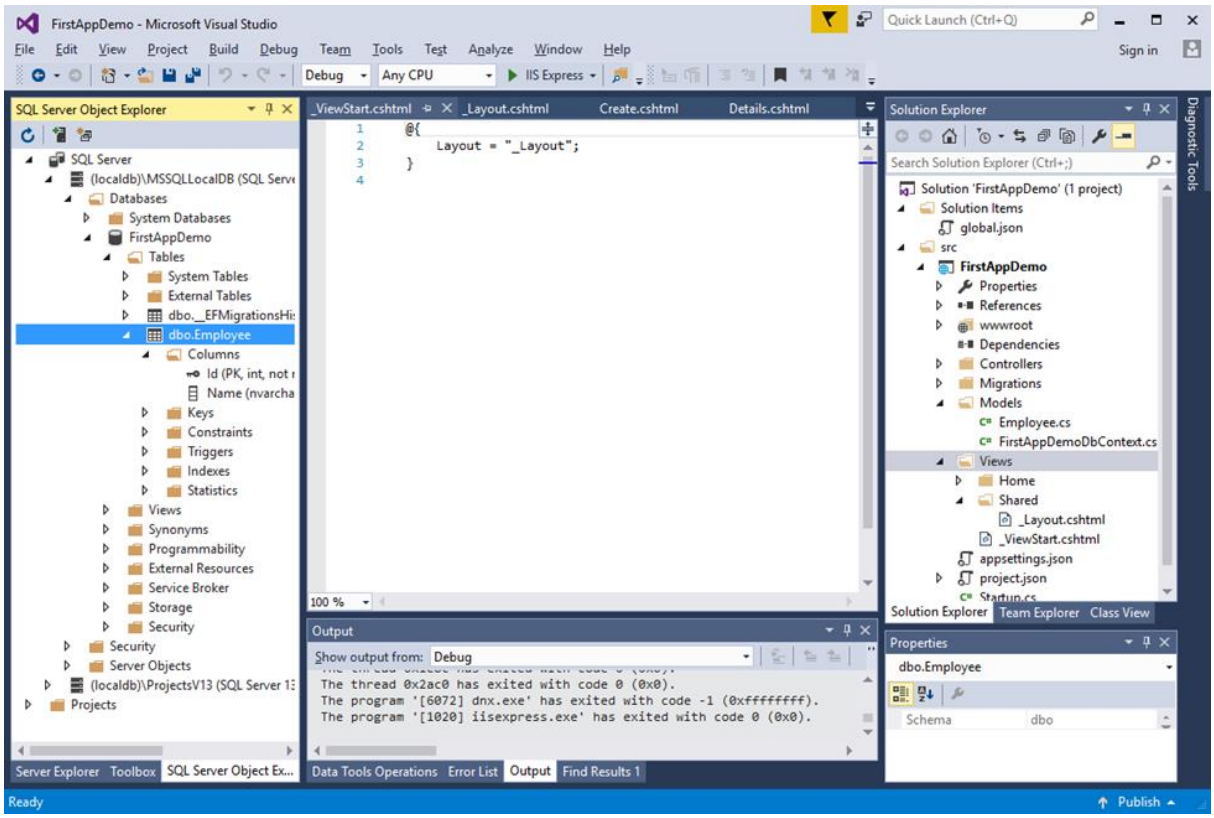
- The code inside the ViewStart file cannot render into the HTML output of a page, but it can be used to remove duplicate code from the code blocks inside the individual views.
- In our example, if we want every view to use the Layout view that we have created in the last chapter, we could put the code to set the Layout view inside a ViewStart instead of having the code inside every view.

Example

Let us take a simple example to see how this works. In our application, we don't want every view to specify that its Layout view is **_Layout.cshtml**. So right-click on the Views folder and select **Add > New Item**.



There is a specific template in ASP.NET MVC for a ViewStart page, so select MVC View Start Page in the middle pane. The most important part here is that this file is named **_ViewStart.cshtml**. Now click on the Add button.



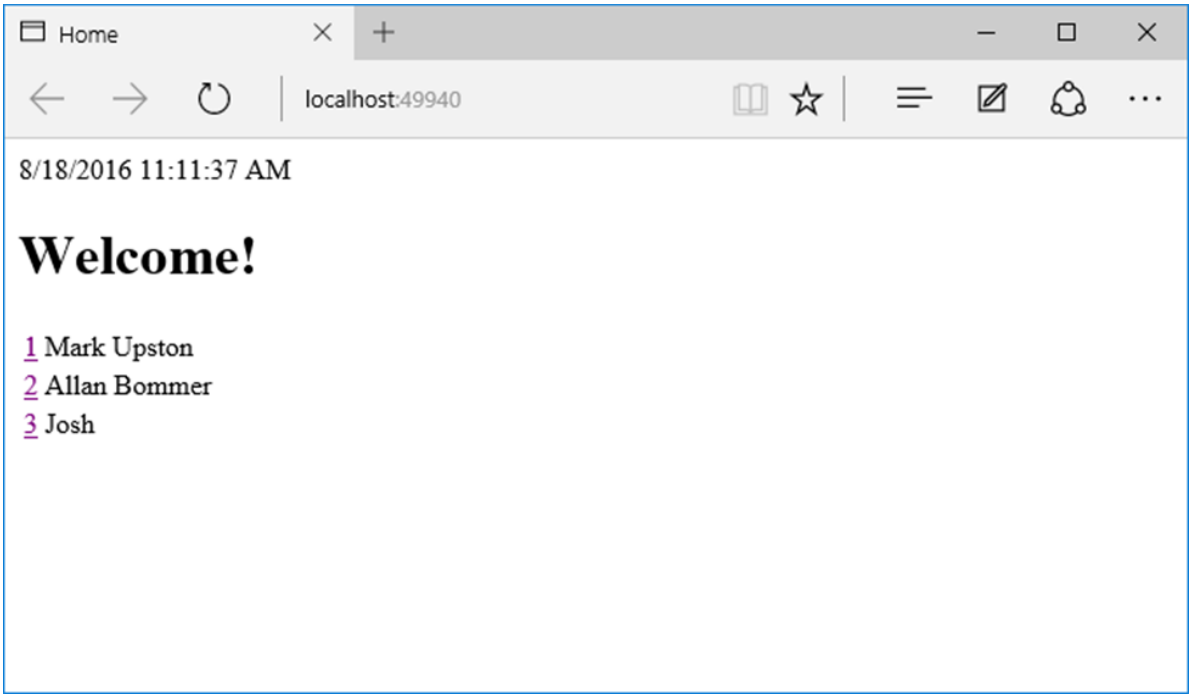
The primary use of a ViewStart file is to set the Layout view.

Let us now go to the Index.cshtml file and cut the Layout line and then add it to the ViewStart file as shown in the following program.

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

- When the MVC framework goes to render a view, it will see if there Is a ViewStart file somewhere in the folder hierarchy.
- We have placed _ViewStart directly into our Views folder. This is going to impact all the views in all the folders that are inside the Views folder, and both the views inside the Home folder, as well as the Shared folder, as well as any other controller folders that we might add in the future.
- If we take ViewStart and place it only in the Home folder, then this little bit of code would only execute when we are rendering one of those views in the Home folder.
- We can even have multiple ViewStart files, so we could have a ViewStart.cshtml here in the Views folder that sets the Layout view for all views.
- But if we wanted to change that default for all of the views just in the Home folder, we could have another ViewStart in the Home folder that sets the layout to something else.

Let us save all the files and run the application.

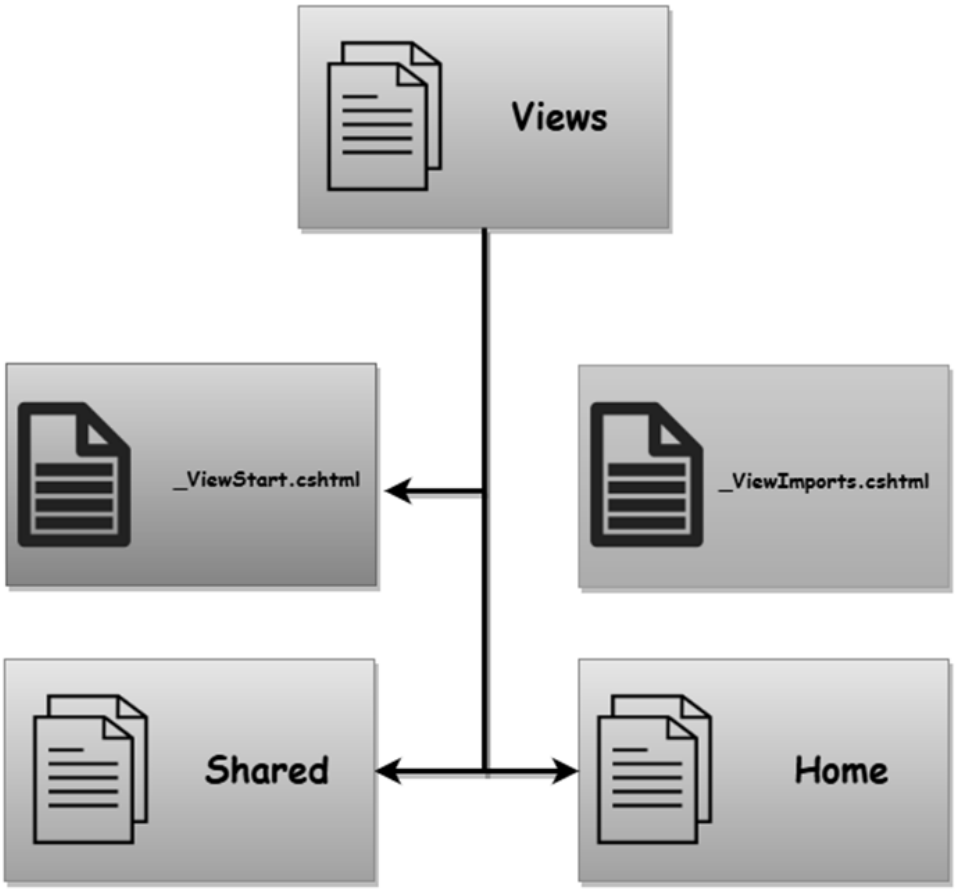


You will see that your home page still renders just the way it did before, and we still have the Layout view in effect.

20. ASP.NET Core — Razor View Import

In this chapter, we will discuss the Razor View Import. In addition to the ViewStart file, there is also a **ViewImports** file that the MVC framework will look for when rendering any view.

Like the ViewStart file, we can drop ViewImports.cshtml into a folder, and the ViewImports file can influence all the views in the folder hierarchy.



- This view is new for this version of MVC, in the previous versions of MVC, we could use an XML configuration file to configure certain aspects of the Razor view engine.
- Those XML files are gone now and we use code instead.
- The ViewImports file is a place where we can write code and place common directives to pull in namespaces that our views need.
- If there are namespaces that we commonly use in our views, we can have **using directives** appear once in our **ViewImports** file instead of having **using directives** in every view or typing out the full namespace of a class.

Example

Let us take a simple example to see how to move our **using directives** into **ViewImports**. Inside the Index view, we have a **using directive** to bring in the namespace **FirstAppDemo.Controllers** as shown in the following program.

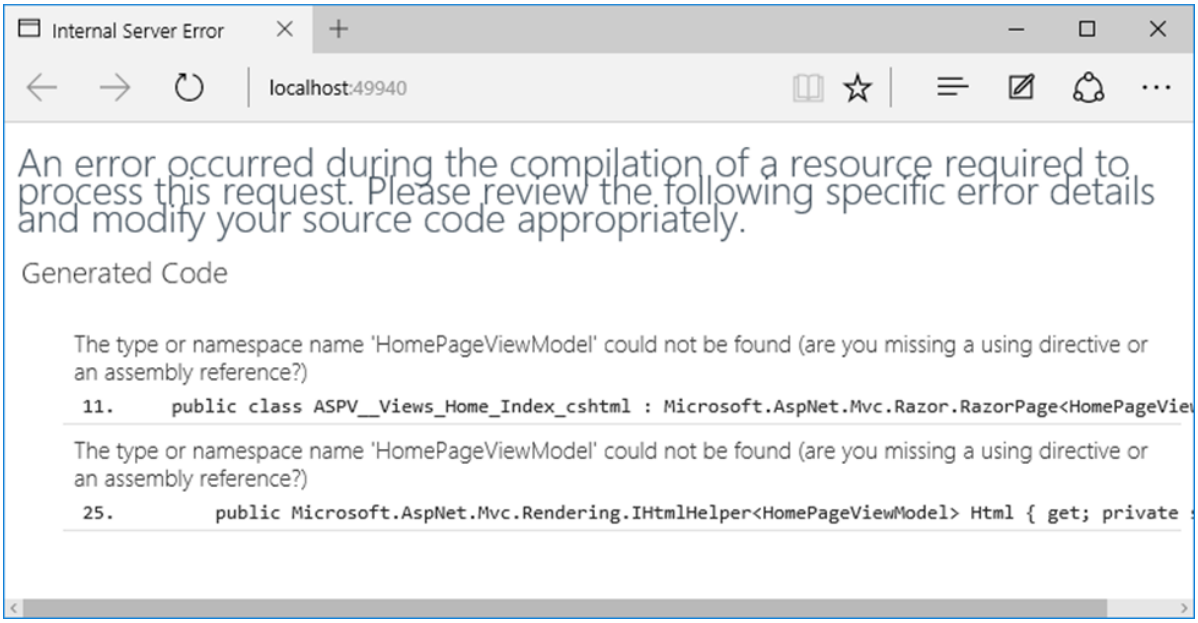
```
@using FirstAppDemo.Controllers
@model HomePageViewModel

@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>
                @Html.ActionLink(employee.Id.ToString(), "Details", new { id =
employee.Id })
            </td>
            <td>@employee.Name</td>
        </tr>
    }
</table>
```

Using directives will allow the code that is generated from the Razor view to compile correctly. Without **using directives**, the C# compiler won't be able to find this Employee type. To see the employee type, let us remove the using directive from the **Index.cshtml** file.

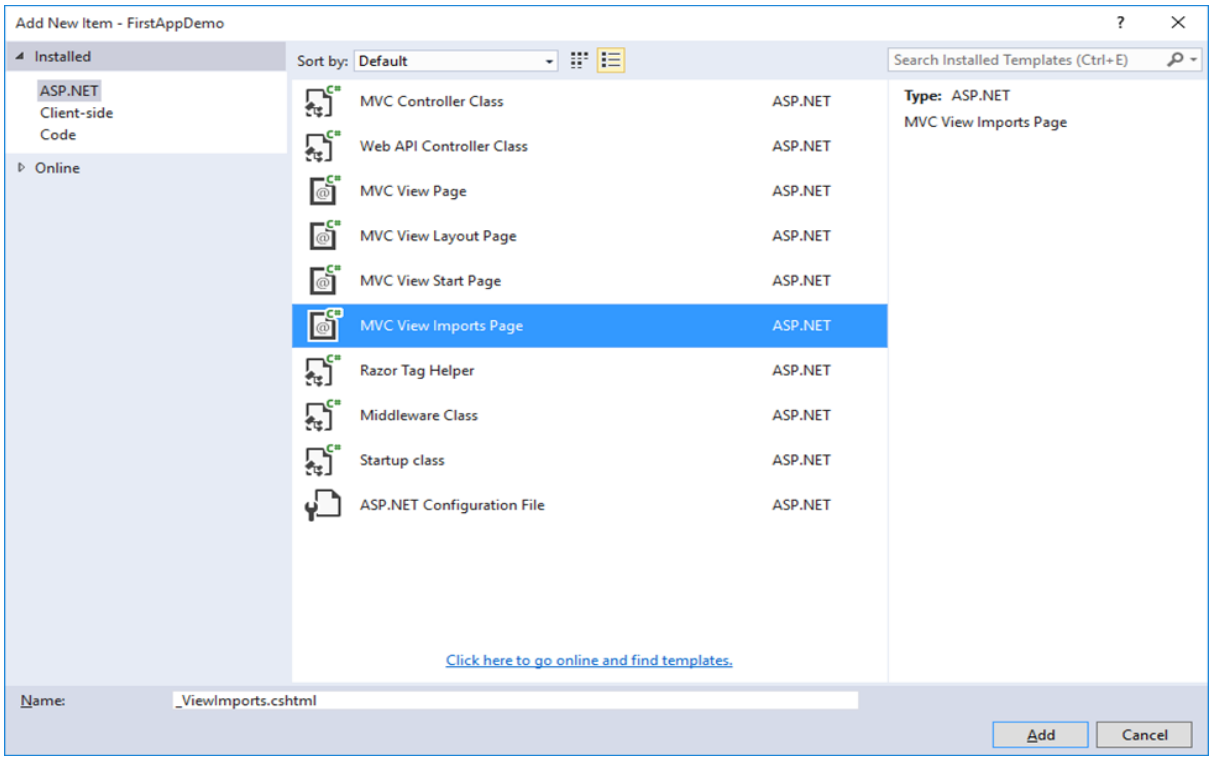
```
@model HomePageViewModel
@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>
```

```
                @Html.ActionLink(employee.Id.ToString(), "Details", new { id =
employee.Id })
            </td>
            <td>@employee.Name</td>
        </tr>
    }
</table>
```

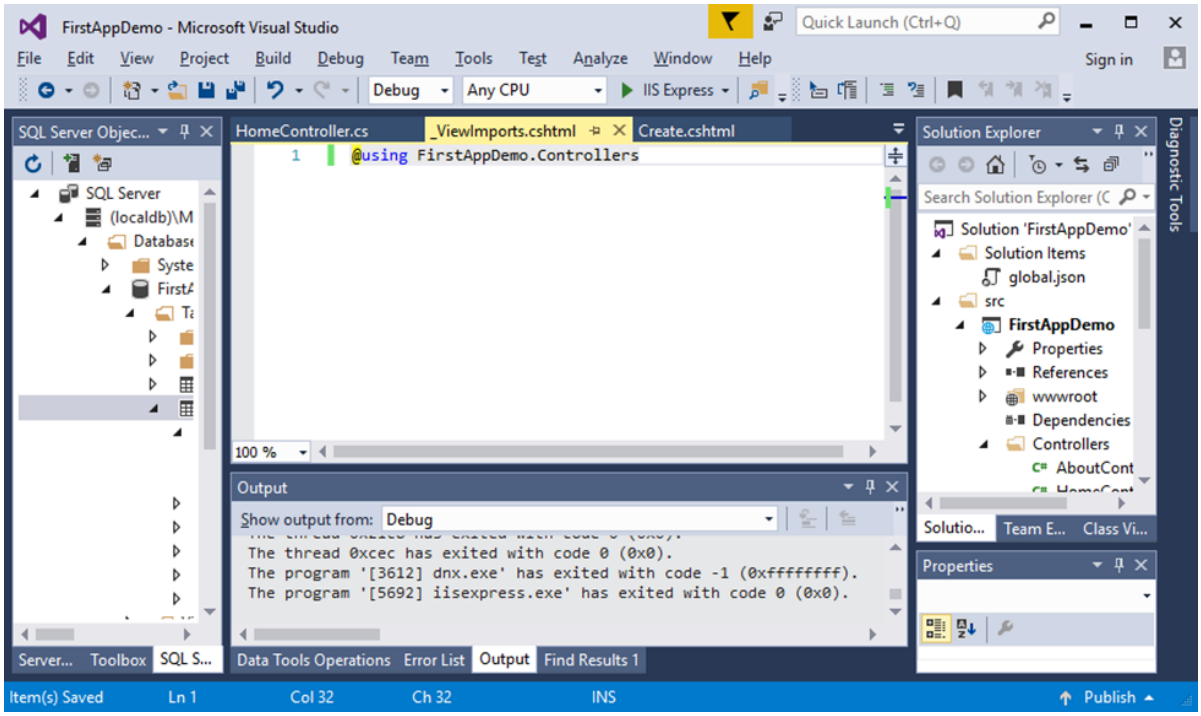
Now, run the application.



You will see one of the errors that states that the type or namespace **HomePageViewModel** could not be found. It might be because several of your views need that same **using directive**. So, instead of placing that inside each view, let us create a View import in the Views folder. This will add **using statements** to every view with just a right-click on the Views folder and selecting Add > New Item.



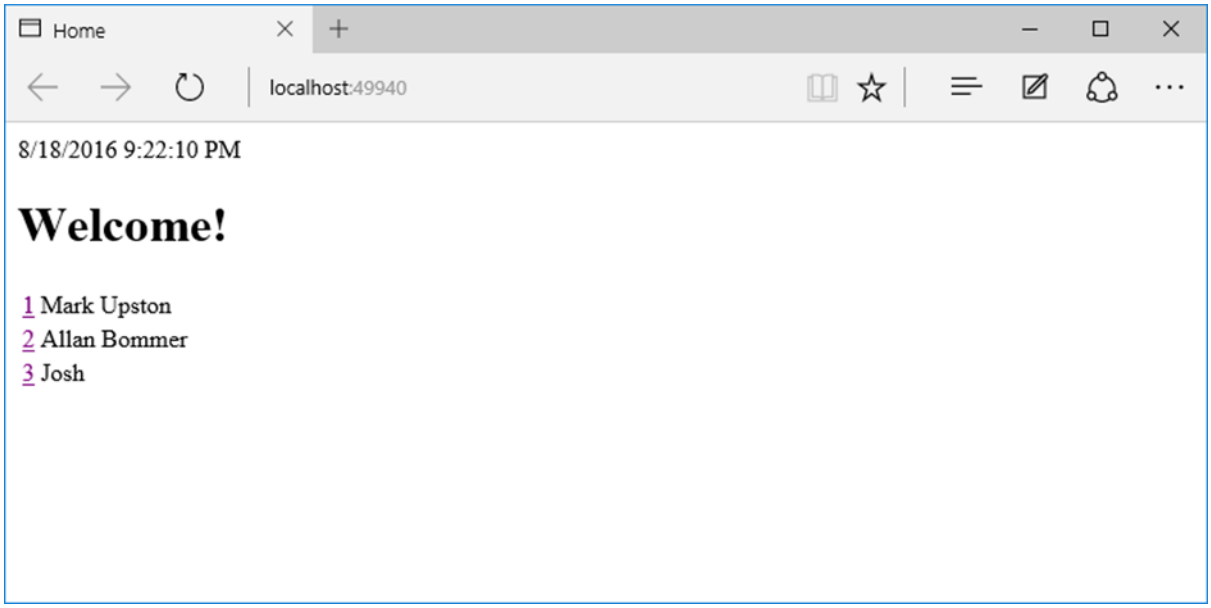
In the middle pane, select the MVC View Imports Page. By default, the name is _ViewImports.cshtml. Just like ViewStart, we cannot use this file to render HTML, so let us click on the Add button.



Now add the **using directive** in this into _ViewImports.cshtml file as shown below.

```
@using FirstAppDemo.Controllers
```

Now, all the views that appear in this folder or any subfolder will be able to use types from FirstAppDemo.Controllers without specifying that exact using statement. Let us run your application again and you can see that the view is working now.



21. ASP.NET Core — Razor Tag Helpers

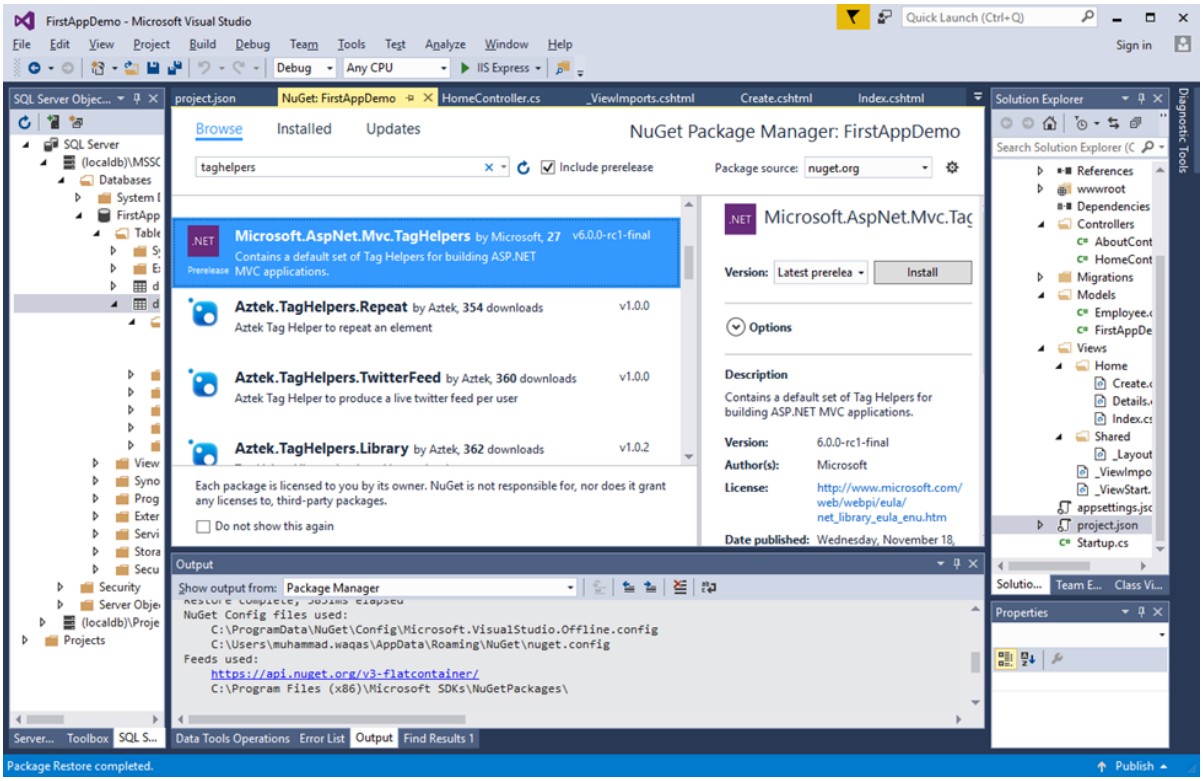
Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag helpers are a new feature and similar to HTML helpers, which help us render HTML.

- There are many built-in Tag Helpers for common tasks, such as creating forms, links, loading assets etc. Tag Helpers are authored in C#, and they target HTML elements based on the element name, the attribute name, or the parent tag.

- For example, the built-in LabelTagHelper can target the HTML <label> element when the LabelTagHelper attributes are applied.

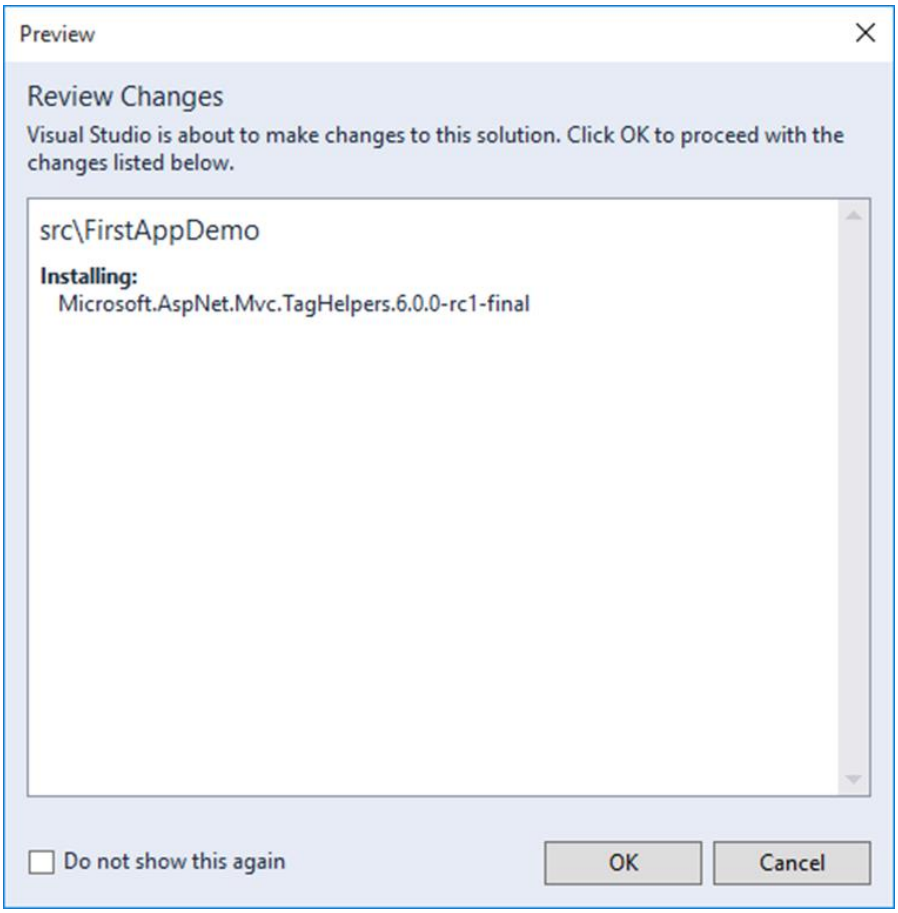
- If you are familiar with HTML Helpers, Tag Helpers reduce the explicit transitions between HTML and C# in Razor views.

In order to use Tag Helpers, we need to install a NuGet library and also add an addTagHelper directive to the view or views that use these tag helpers. Let us right-click on your project in the Solution Explorer and select Manage NuGet Packages...

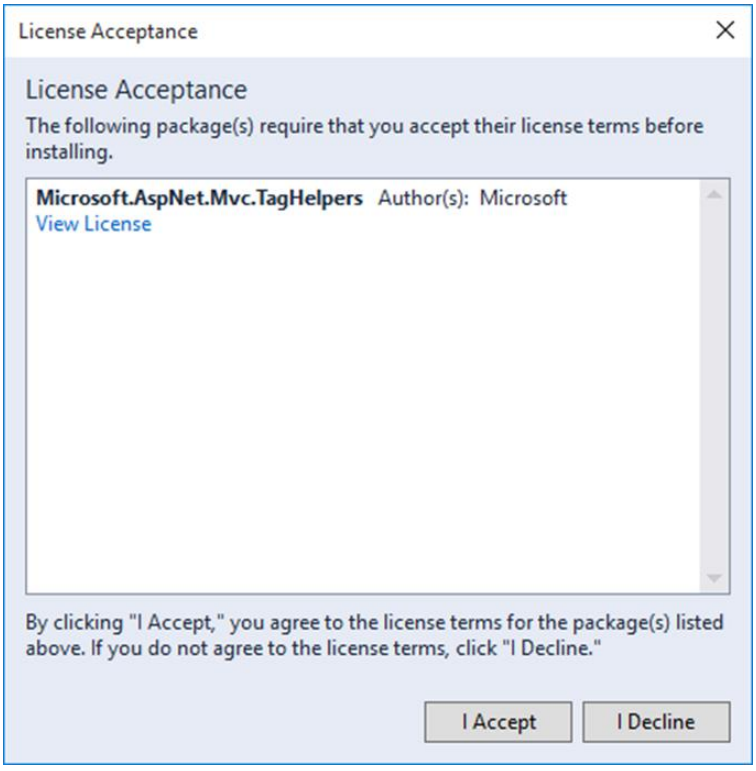


Search for **Microsoft.AspNet.Mvc.TagHelpers** and click the Install button.

You will receive the following Preview dialog box.



Click the OK button.



Click the **I Accept** button. Once the Microsoft.AspNet.Mvc.TagHelpers is installed, go to the project.json file.

```
{
  "version": "1.0.0-*",
  "compilationOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Diagnostics": "1.0.0-rc1-final",
    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "EntityFramework.Commands": "7.0.0-rc1-final",
    "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel",
    "ef": "EntityFramework.Commands"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
  "publishExclude": [
    "**.user",
    "**.vspscc"
  ]
}
```

In the dependencies section, you will see that "**Microsoft.AspNet.Mvc.TagHelpers**": "**6.0.0-rc1-final**" is added.

- Now anybody can author a tag helper, so if you can think of a tag helper that you need, you can write your own tag helper.
- You can place it right inside your application project, but you need to tell the Razor view engine about the tag helper
- By default, they are not just rendered down to the client, even though these tag helpers look like they blend into the HTML.
- Razor will call into some code to process a tag helper; it can remove itself from the HTML and it can also add additional HTML.
- There are many wonderful things that you can do with a tag helper, but you need to register your tag helpers with Razor, even the Microsoft tag helpers, in order for Razor to be able to spot these tag helpers in the markup and to be able to call into the code that processes the tag helper.
- The directive to do that is addTagHelper, and you can place this into an individual view, or if you plan on using tag helpers throughout the application, you can use addTagHelper inside the ViewImports file as shown below.

```
@using FirstAppDemo.Controllers
@addTagHelper "*, Microsoft.AspNet.Mvc.TagHelpers"
```

The syntax to register all the tag helpers that are in an assembly is to use asterisk **comma** (*,) and then the assembly name, **Microsoft.AspNet.Mvc.TagHelpers**. Because the first piece here is a type name, this is where we could list a specific tag helper if you only wanted to use one.

But if you just wanted to take all the tag helpers that are in this assembly, you can just use the **asterisk**(*). There are many tag helpers available in the tag helper library. Let us have a look at the Index view.

```
@model HomePageViewModel


@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>
```

```
                @Html.ActionLink(employee.Id.ToString(), "Details", new { id =
employee.Id })
            </td>
            <td>@employee.Name</td>
        </tr>
    }
</table>
```

We already have an HTML helper using the **ActionLink** to generate an anchor tag that will point to a URL that allows us to get to the details of an employee.

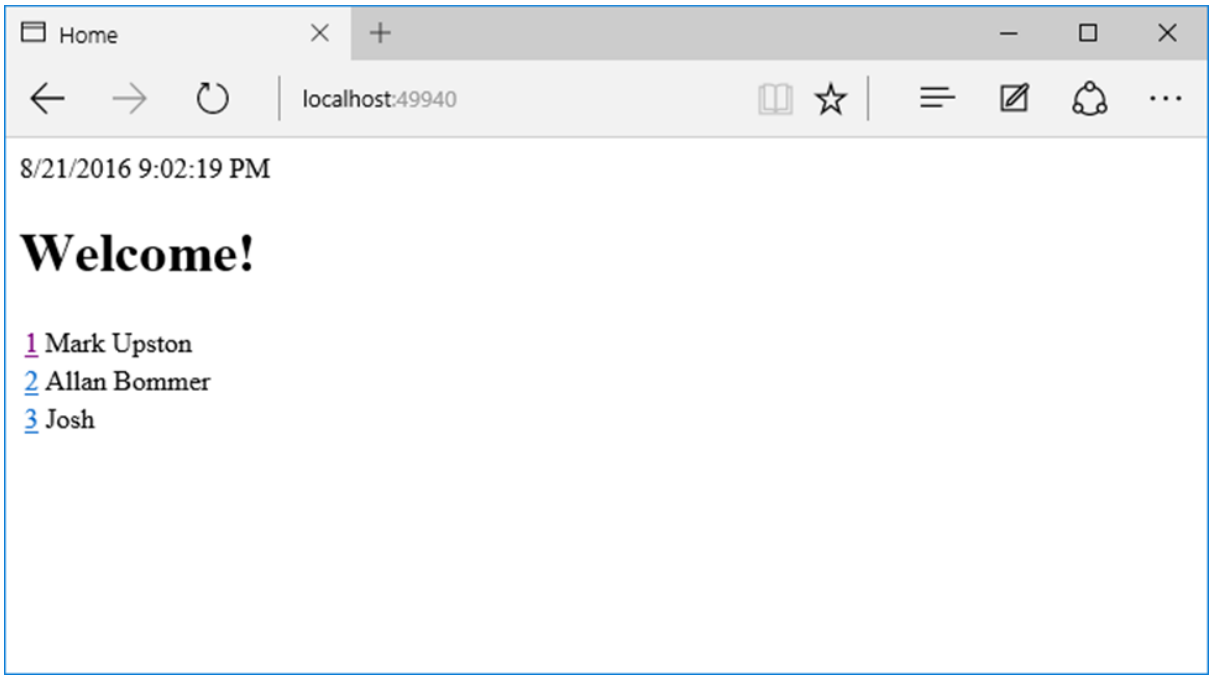
Let us first add the Details action in the home controller as shown in the following program.

```
public IActionResult Details(int id)
{
    var context = new FirstAppDemoDbContext();
    SQLEmployeeData sqlData = new SQLEmployeeData(context);
    var model = sqlData.Get(id);
    if (model == null)
    {
        return RedirectToAction("Index");
    }
    return View(model);
}
```

Now we need to add a view for the Details action. Let us create a new view in the Views > Home folder and call it Details.cshtml and add the following code.

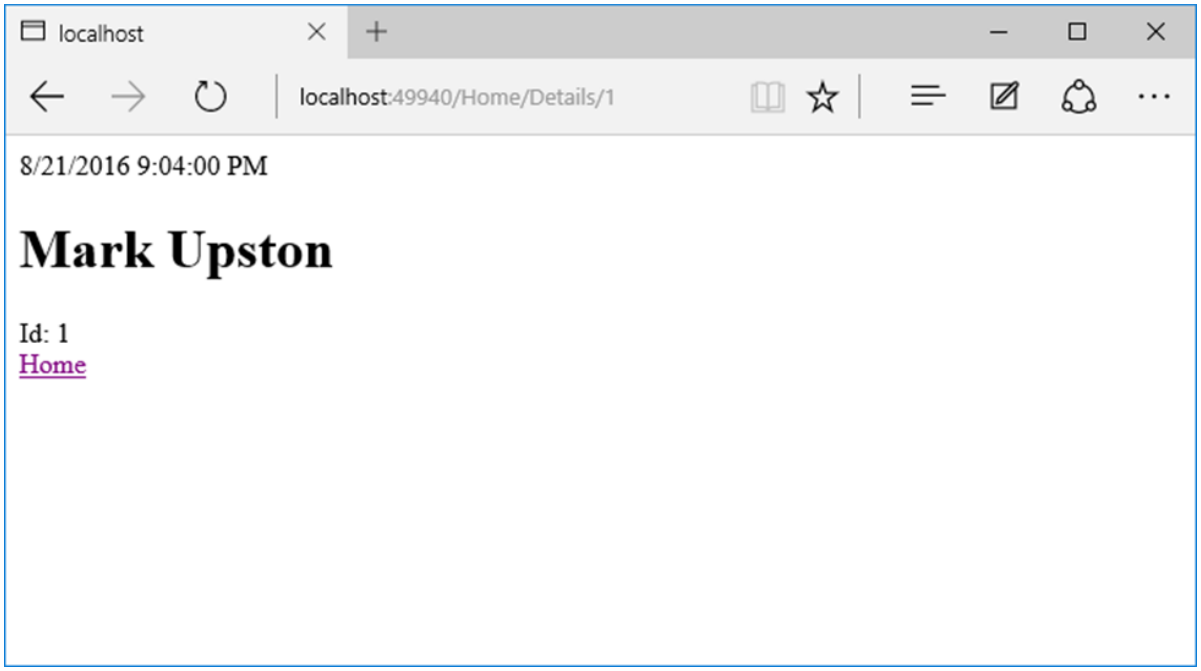
```
@model FirstAppDemo.Models.Employee
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>@Model.Name</title>
</head>
<body>
    <h1>@Model.Name</h1>
    <div>Id: @Model.Id</div>
    <div>
        @Html.ActionLink("Home", "Index")
    </div>
</body>
</html>
```

Let us now run the application.



When you click on the ID of an employee then it will get you to the details view.

Let us click the first employee ID.



Now to use tag helper for this, let us add the following line in the index.cshtml file and remove the HTML helper.

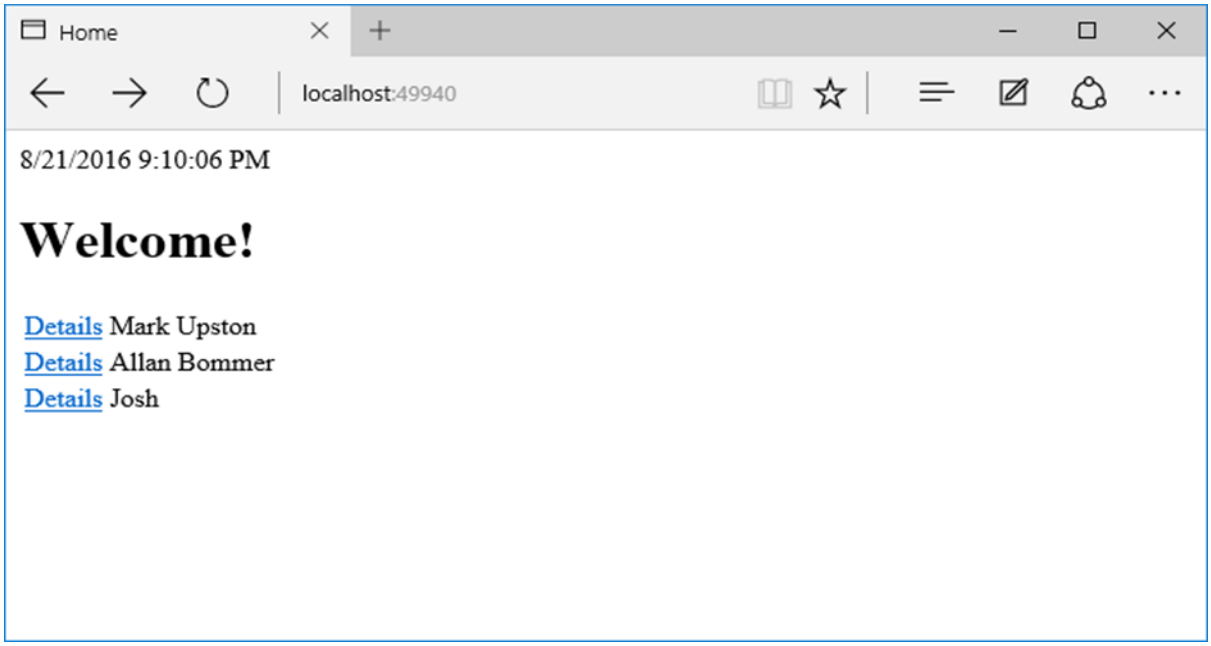
```
<a asp-action="Details" asp-rout-id="@employee.Id" >Details</a>
```

The **asp-action="Details"** is the name of the action that we want to get to. If there is any parameter that you want to be passed along, you can use the asp-route tag helper and here we want to include ID as parameter so we can use asp-route-Id taghelper.

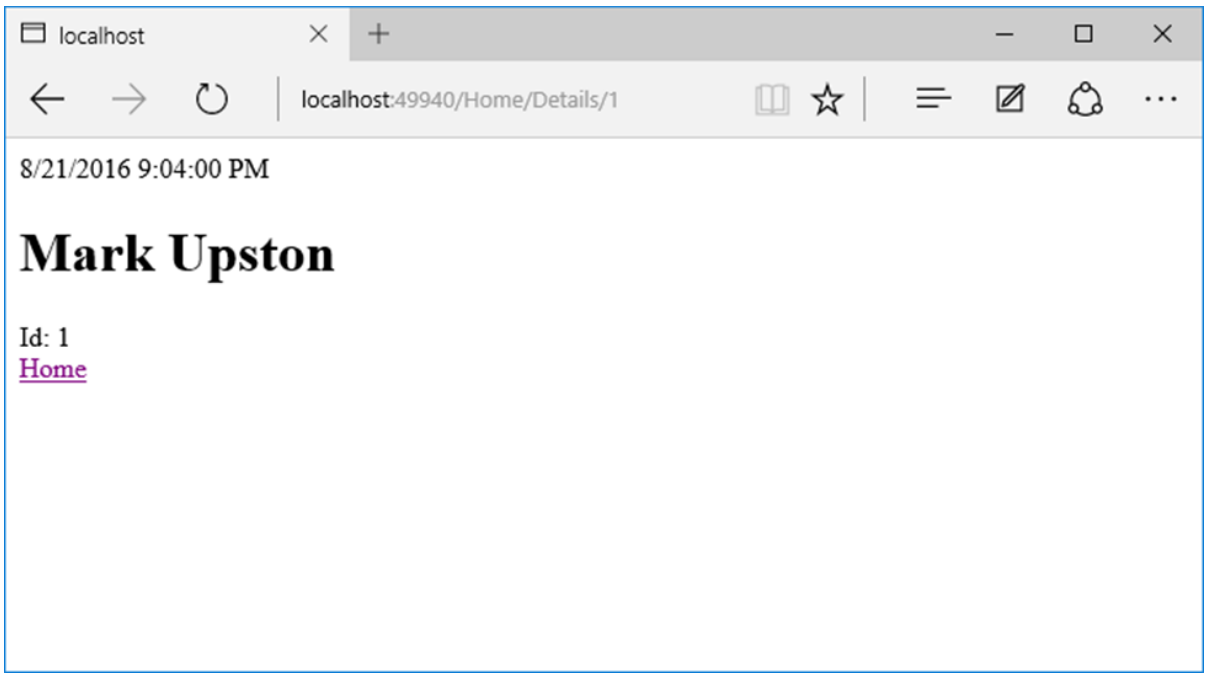
The following is the complete implantation of index.cshtml file.

```
@model HomePageViewModel


@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>
                <a asp-action="Details" asp-route-id="@employee.Id"
>Details</a>
            </td>
            <td>@employee.Name</td>
        </tr>
    }
</table>
```

Let us run your application again. After you run the application, you will see the following page.



Previously, we were displaying the ID as the linking text, but now we are showing the text Details. Now, we click on the details and are creating the correct URL now using the tag helpers instead of the HTML helpers.



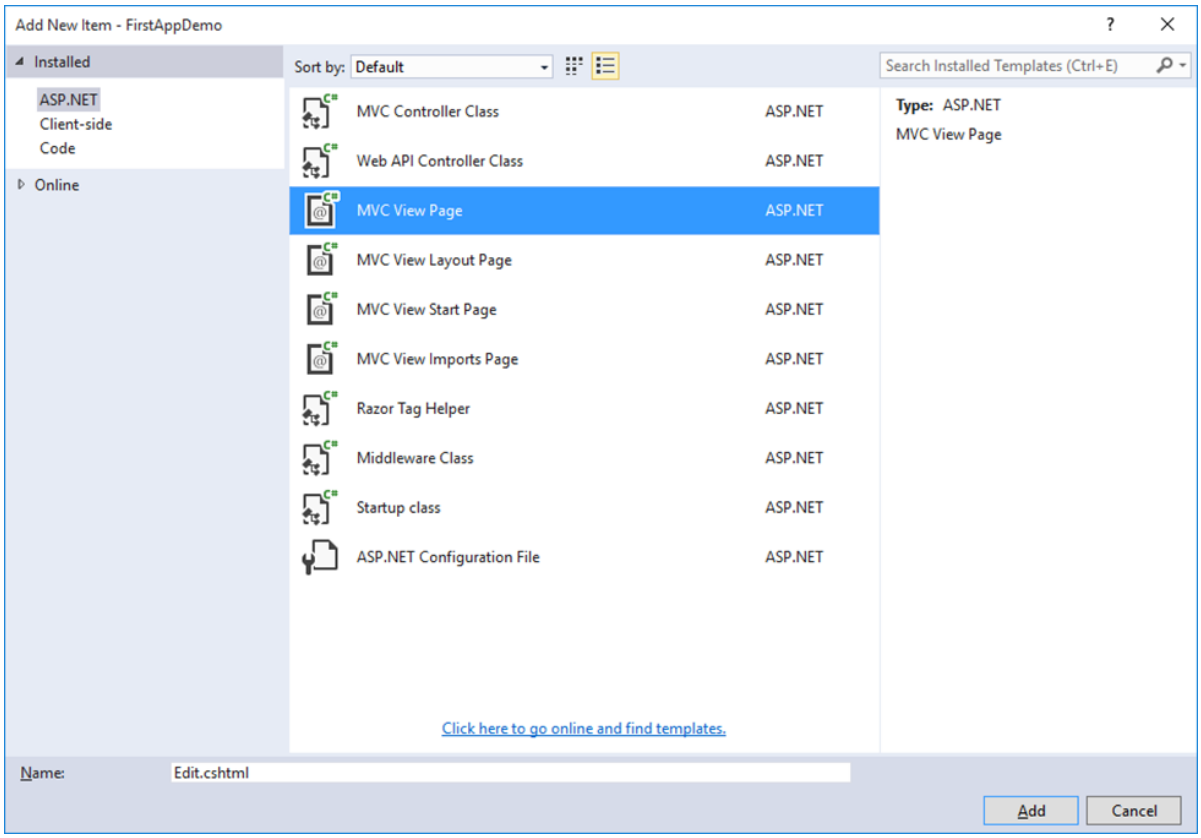
Whether you choose to use **HTML helpers** or **tag helpers**, it's really a matter of personal preference. Many developers find tag helpers to be easier to author and maintain.

22. ASP.NET Core — Razor Edit Form

In this chapter, we will continue discussing the tag helpers. We will also add a new feature in our application and give it the ability to edit the details of an existing employee. We will start by adding a link on the side of each employee that will go to an Edit action on the HomeController.

```
@model HomePageViewModel


@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>@employee.Name</td>
            <td>
                <a asp-controller="Home" asp-action="Details" asp-route-
id="@employee.Id">Details</a>
                <a asp-controller="Home" asp-action="Edit" asp-route-
id="@employee.Id">Edit</a>
            </td>
        </tr>
    }
</table>
```

We don't have the Edit action yet, but we will need an employee ID that we can edit. So let us first create a new view by right-clicking on the **Views → Home** folder and select **Add > New Items.**



In the middle pane, select the MVC View Page; call the page Edit.cshtml. Now, click on the Add button.

Add the following code in the **Edit.cshtml** file.

```
@model Employee
@{
    ViewBag.Title = $"Edit {Model.Name}";
}
<h1>Edit @Model.Name</h1>

<form asp-action="Edit" method="post">
    <div>
        <label asp-for="Name"></label>
        <input asp-for="Name" />
        <span asp-validation-for="Name"></span>
    </div>
    <div>
        <input type="submit" value="Save" />
    </div>
</form>
```

For the title of this page, we can say that we want to edit and then provide the employee name.

- The dollar sign in front of **Edit** will allow the runtime to replace Model.Name with a value that is in that property like employee name.

- Inside the form tag, we can use tag helpers like asp-action and asp-controller. so that when the user submits this form it goes directly to a specific controller action.

- In this case, we want to go to the Edit action on the same controller and we want to explicitly say that for the method on this form, it should be using an HttpPost.

- The default method for a form is a GET, and we do not want to edit an employee using a GET operation.

- In the label tag, we have used asp-for tag helper which says that this is a label for the Name property of the model. This tag helper can set up the Html.For attribute to have the correct value and to set the inner text of this label so that it actually displays what we want, like employee name.

Let us go to the HomeController class and add **Edit** action that returns the view that gives the user a form to edit an employee and then we will need a second Edit action that will respond to an HttpPost as shown below.

```
[HttpGet]
public IActionResult Edit(int id)
{
    var context = new FirstAppDemoDbContext();
    SQLEmployeeData sqlData = new SQLEmployeeData(context);
    var model = sqlData.Get(id);
    if (model == null)
    {
        return RedirectToAction("Index");
    }
    return View(model);
}
```

First, we need an edit action that will respond to a GET request. It will take an employee ID. The code here will be similar to the code that we have in the Details action. We will first extract the data of the employee that the user wants to edit. We also need to make sure that the employee actually exists. If it doesn't exist, we will redirect the user back to the Index view. But when an employee exists, we will render the Edit view.

We also need to respond to the HttpPost that the form will send.

Let us add a new class in the HomeController.cs file as shown in the following program.

```
public class EmployeeEditViewModel
{
    [Required, MaxLength(80)]
    public string Name { get; set; }
}
```

In the Edit Action which will respond to the HttpPost will take an EmployeeEditViewModel, but not an employee itself, because we only want to capture items that are in form in the Edit.cshtml file.

The following is the implementation of the Edit action.

```
[HttpPost]
public IActionResult Edit(int id, EmployeeEditViewModel input)
{
    var context = new FirstAppDemoDbContext();
    SQLEmployeeData sqlData = new SQLEmployeeData(context);
    var employee = sqlData.Get(id);
    if (employee != null && ModelState.IsValid)
    {
        employee.Name = input.Name;
        context.SaveChanges();

        return RedirectToAction("Details", new { id = employee.Id });
    }
    return View(employee);
}
```

The edit form should always be delivered from an URL that has an ID in the URL according to our routing rules, something like **/home/edit/1**.

- The form is always going to post back to that same URL, /home/edit/1.
- The MVC framework will be able to pull that ID out of the URL and pass it as a parameter.
- We always need to check if the ModelState is valid and also make sure that this employee is in the database and it is not null before we perform an update operation in the database.
- If none of that is true, we will return a view and allow the user to try again. Although in a real application with concurrent users, if the employee is null, it could be because the employee details were deleted by someone.

- If that employee doesn't exist, tell the user that the employee doesn't exist.
- Otherwise, check the ModelState. If the ModelState is invalid, then return a view. This allows to fix the edit and make the ModelState valid.
- Copy the name from the Input view model to the employee retrieved from the database and save the changes. The SaveChagnes() method is going to flush all those changes to the database.

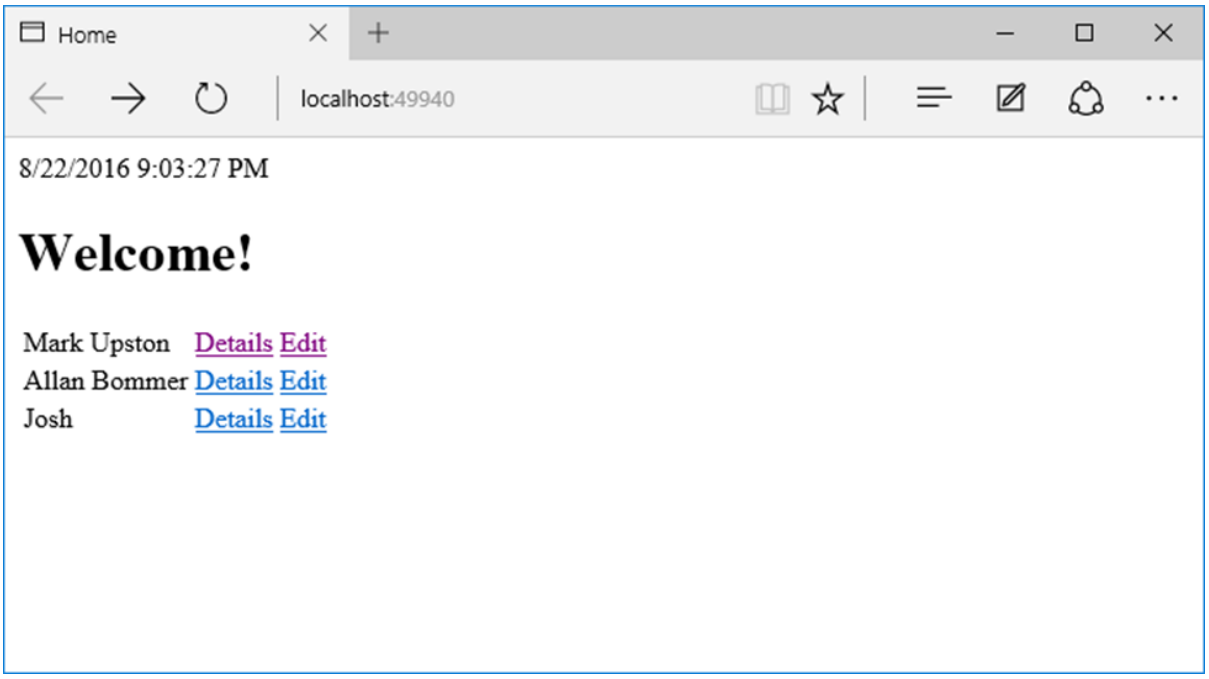
The following is the complete implementation of the HomeController.

```
using Microsoft.AspNet.Mvc;
using FirstAppDemo.ViewModels;
using FirstAppDemo.Services;
using FirstAppDemo.Entities;
using FirstAppDemo.Models;
using System.Collections.Generic;
using System.Linq;
using System.ComponentModel.DataAnnotations;

namespace FirstAppDemo.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            var model = new HomePageViewModel();
            using (var context = new FirstAppDemoDbContext())
            {
                SQLEmployeeData sqlData = new SQLEmployeeData(context);
                model.Employees = sqlData.GetAll();
            }

            return View(model);
        }

        public IActionResult Details(int id)
        {
            var context = new FirstAppDemoDbContext();
            SQLEmployeeData sqlData = new SQLEmployeeData(context);
            var model = sqlData.Get(id);
```

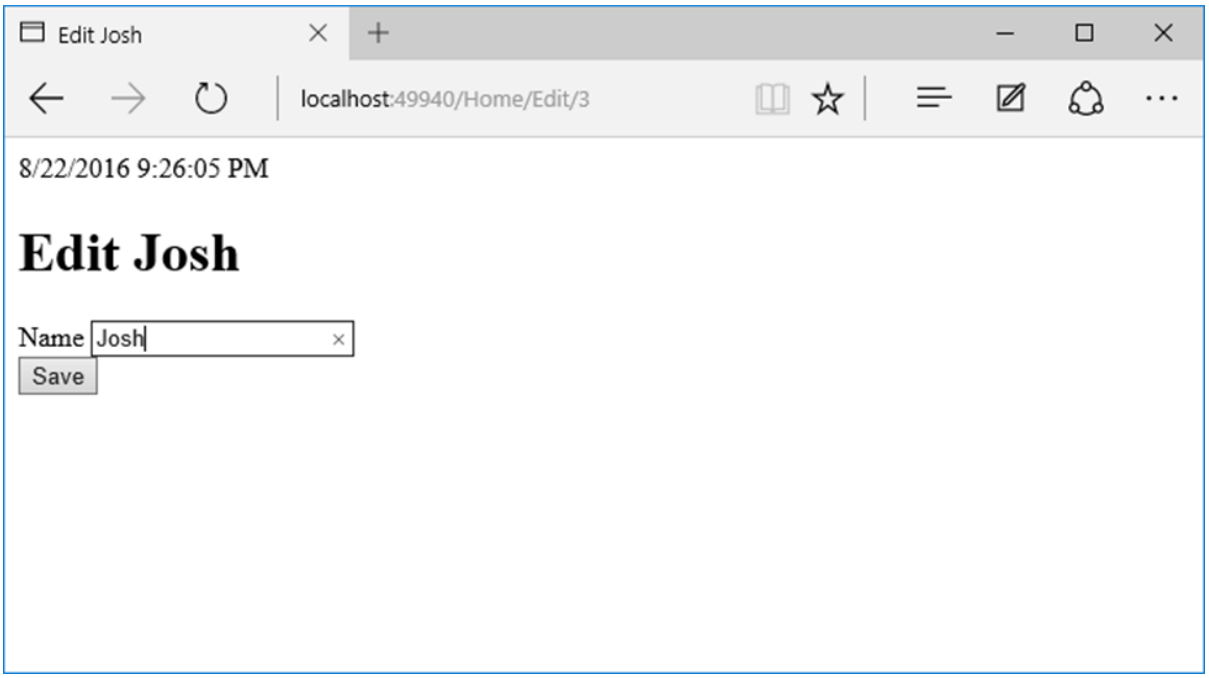
```
            if (model == null)

            {
                return RedirectToAction("Index");
            }
            return View(model);
        }
        [HttpGet]
        public IActionResult Edit(int id)
        {
            var context = new FirstAppDemoDbContext();
            SQLEmployeeData sqlData = new SQLEmployeeData(context);
            var model = sqlData.Get(id);
            if (model == null)
            {
                return RedirectToAction("Index");
            }
            return View(model);
        }

        [HttpPost]
        public IActionResult Edit(int id, EmployeeEditViewModel input)
        {
            var context = new FirstAppDemoDbContext();
            SQLEmployeeData sqlData = new SQLEmployeeData(context);
            var employee = sqlData.Get(id);
            if (employee != null && ModelState.IsValid)
            {
                employee.Name = input.Name;
                context.SaveChanges();

                return RedirectToAction("Details", new { id = employee.Id });
            }
            return View(employee);
        }
    }
```

```
    public class SQLEmployeeData
    {
        private FirstAppDemoDbContext _context { get; set; }
        public SQLEmployeeData(FirstAppDemoDbContext context)
        {
            _context = context;
        }

        public void Add(Employee emp)
        {
            _context.Add(emp);
            _context.SaveChanges();
        }
        public Employee Get(int ID)
        {
            return _context.Employees.FirstOrDefault(e => e.Id == ID);
        }
        public IEnumerable<Employee> GetAll()
        {
            return _context.Employees.ToList<Employee>();
        }
    }
    public class HomePageViewModel
    {
        public IEnumerable<Employee> Employees { get; set; }
    }

    public class EmployeeEditViewModel
    {
        [Required, MaxLength(80)]
        public string Name { get; set; }
    }
}
```

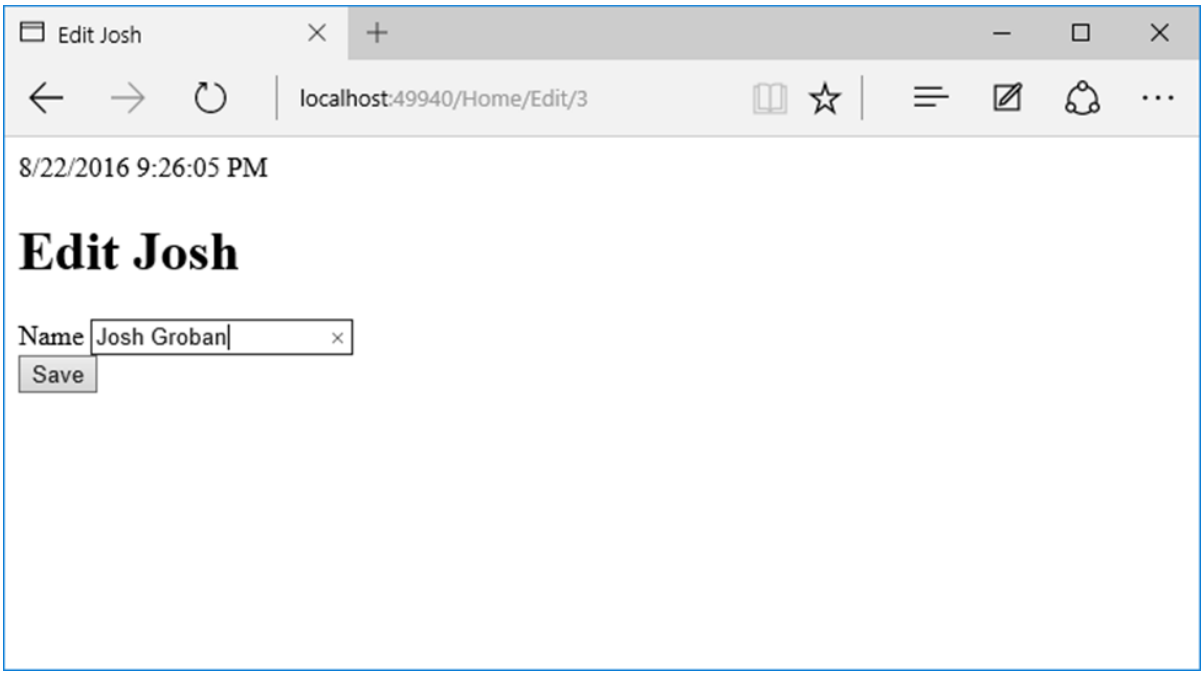
Let us compile the program and run the application.



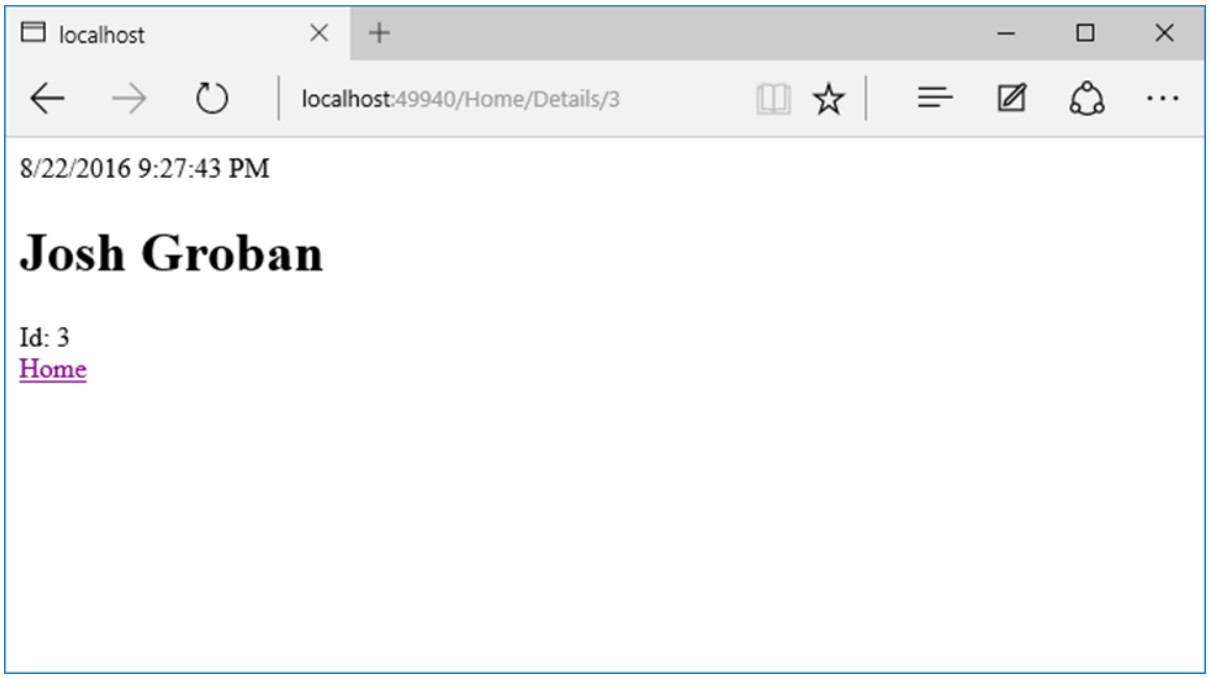
We now have an Edit link available; let us edit the details of Josh by clicking on the Edit link.



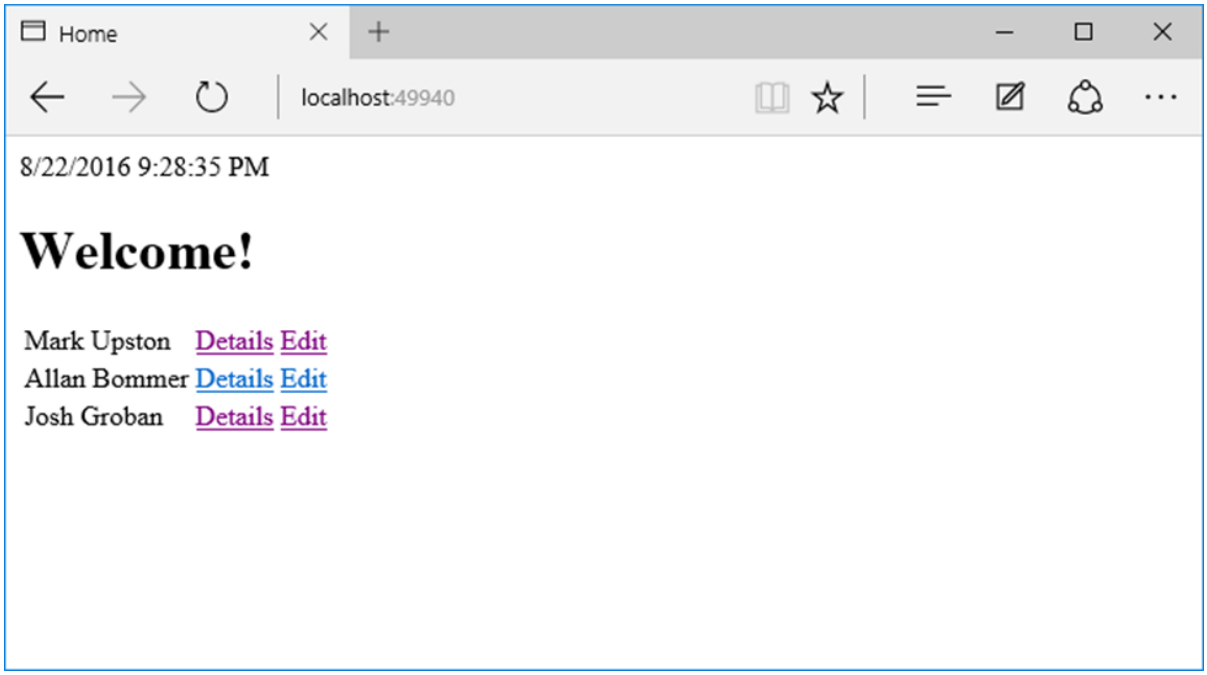
Let us change the name to Josh Groban.



Click the Save button.



You can see that the name has been changed to Josh Groban as in the above screenshot. Let us now click on the Home link.



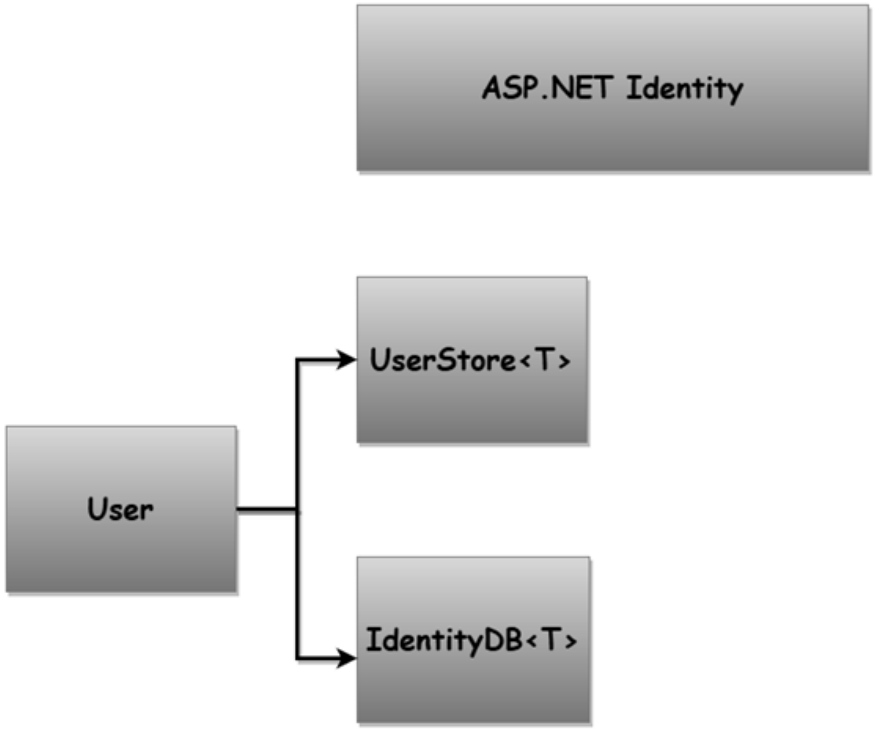
On the home page, you will now see the updated name.

23. ASP.NET Core — Identity Overview

In this chapter, we will discuss the ASP.NET Core Identity framework in brief. ASP.NET Core Identity framework is used to implement forms authentication. There are many options to choose from for identifying your users including Windows Authentication and all of the third-party identity providers like Google, Microsoft, Facebook, and GitHub etc.

- The Identity framework is another dependency that we will add to our application in the project.js file.
- This framework allows us to add features where users can register and log in with a local password.
- The framework also supports two-factor authentication, third-party identity providers and other features.
- We are going to focus on the scenarios where a user can register and log in and log out.

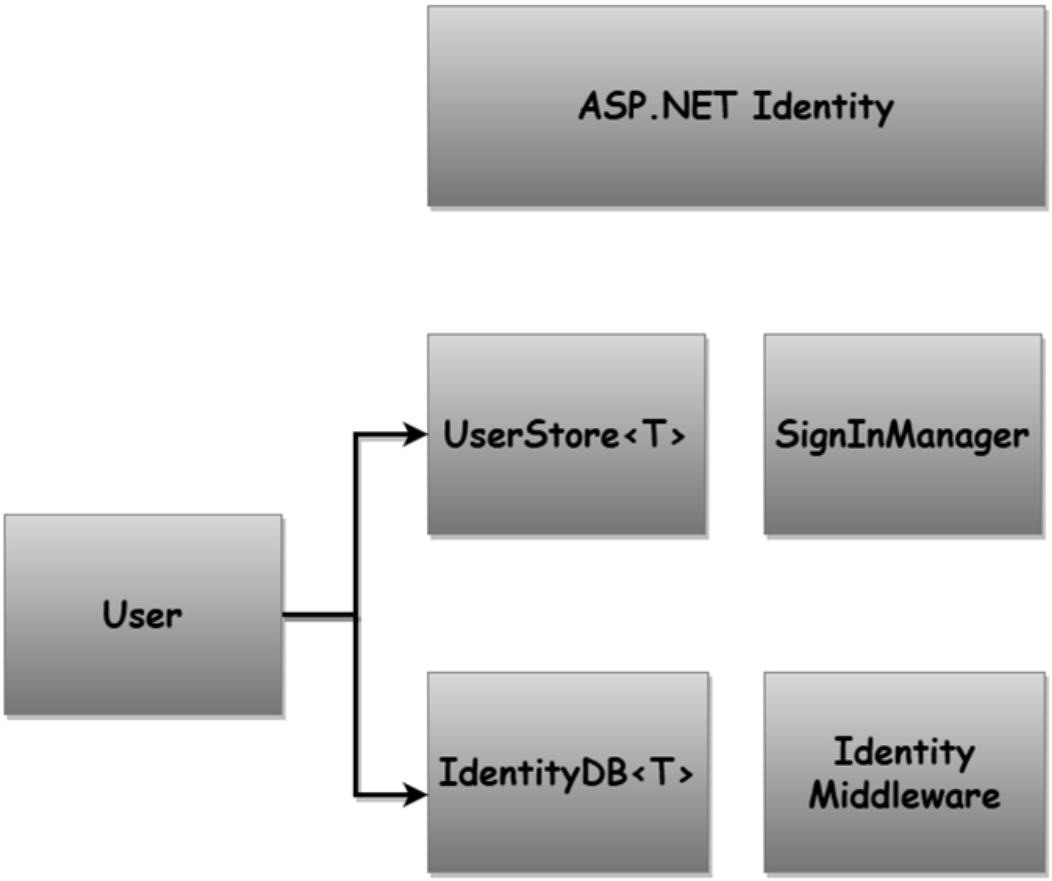
To do this, we need to create a User entity and this class will inherit from a base class in the Identity framework, and the base class gives us our standard user properties like username and email address.



- We can include as many additional properties as we want on this class to store information about our users.
- We need to take this User class and plug it into a UserStore class provided by the Identity framework.

- The UserStore is the class that our code will talk to create users and validate user passwords.
- Ultimately, a UserStore will talk to a database. The Identity framework supports the Entity Framework and all of the databases that can work with the Entity Framework.
- But you can implement your own UserStore to work with any data source.
- In order to work with the Entity Framework properly, our User class will also plug into an IdentityDb class.
- This is a class that uses an Entity Framework DBContext to do the actual database work.
- We will need to include this IdentityDb into our application by having our existing DataContext class inherit from the IdentityDb instead of the Entity Framework's DBContext.
- It is the IdentityDb and the UserStore that work together to store user information and validate user passwords, the hashed passwords that are in the database.

There are two pieces of the ASP.NET Core Identity Framework that we need to know



SignInManager

This is one of the two pieces of the Identity framework:

- As the name implies, the **SignInManager** can sign in a user once we validate the password.
- We can also use this manager to sign a user out.
- With forms authentication, the sign in and the sign out are done by managing a cookie.
- When we tell the SignInManager to sign a user in, the manager issues a cookie to the user's browser, and the browser will send this cookie on every subsequent request. It helps us identify that user.

Identity Middleware

This is the second piece of the framework:

- Reading the cookie sent by the SignInManager and identifying the user, this happens in the final piece of the framework, the Identity Middleware.
- We will need to configure this middleware into our application pipeline to process the cookie set by the SignInManager. We will also see some other features of this middleware in the next few chapters.

24. ASP.NET Core — Authorize Attribute

In this chapter, we will discuss the Authorize Attribute. So far in our application, we have allowed anonymous users to do anything. They can edit employee details, and view details, but we don't have the functionality to create a new employee. Let us first add the create feature and then we will restrict the user access using Authorize attribute.

We need to start by creating a new MVC View page inside **Views > Home** folder and call it Create.cshtml and then add the following code.

```
@model Employee
@{
    ViewBag.Title = "Create";
}
<h1>Create</h1>

@using (Html.BeginForm())
{
    <div>
        @Html.LabelFor(m => m.Name)
        @Html.EditorFor(m => m.Name)
        @Html.ValidationMessageFor(m => m.Name)
    </div>
    <div>
        <input type="submit" value="Save" />
    </div>
}
```

We will now add the **action method** in the HomeController for both POST and GET as shown in the following program.

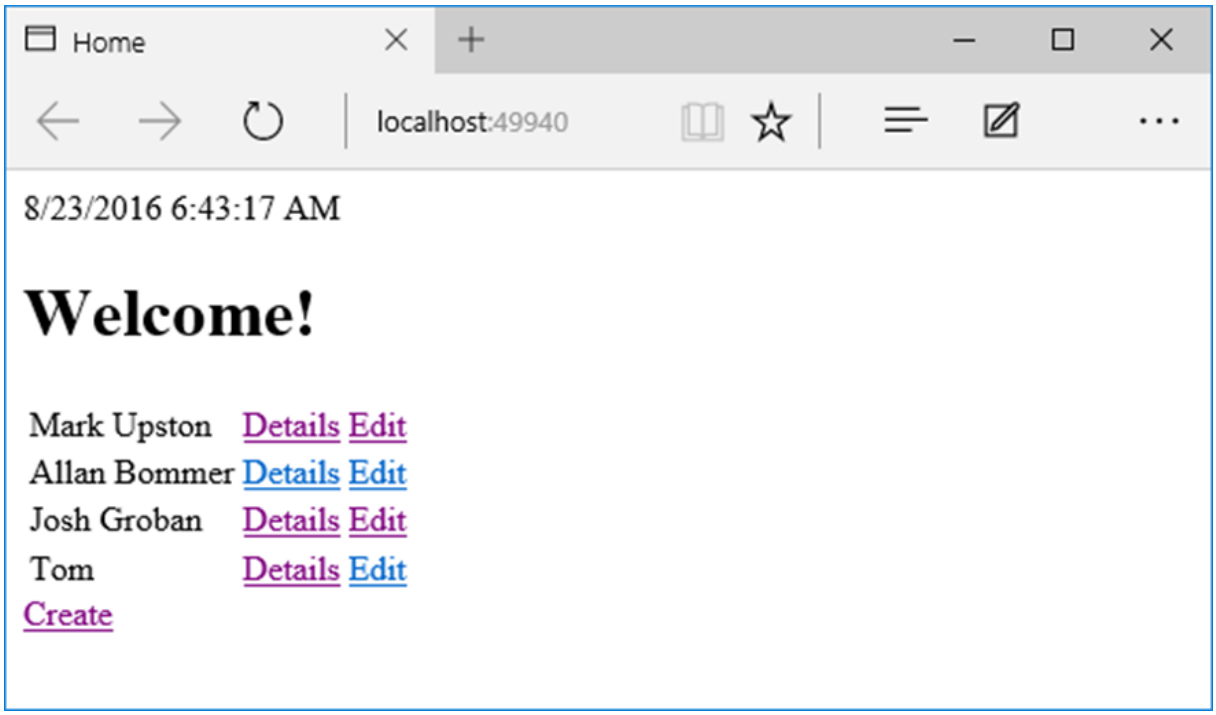
```
[HttpGet]
public ViewResult Create()
{
    return View();
}

[HttpPost]
public IActionResult Create(EmployeeEditViewModel model)
{
    if (ModelState.IsValid)
```

```
    {
        var employee = new Employee();
        employee.Name = model.Name;

        var context = new FirstAppDemoDbContext();
        SQLEmployeeData sqlData = new SQLEmployeeData(context);
        sqlData.Add(employee);

        return RedirectToAction("Details", new { id = employee.Id });
    }
    return View();
}
```

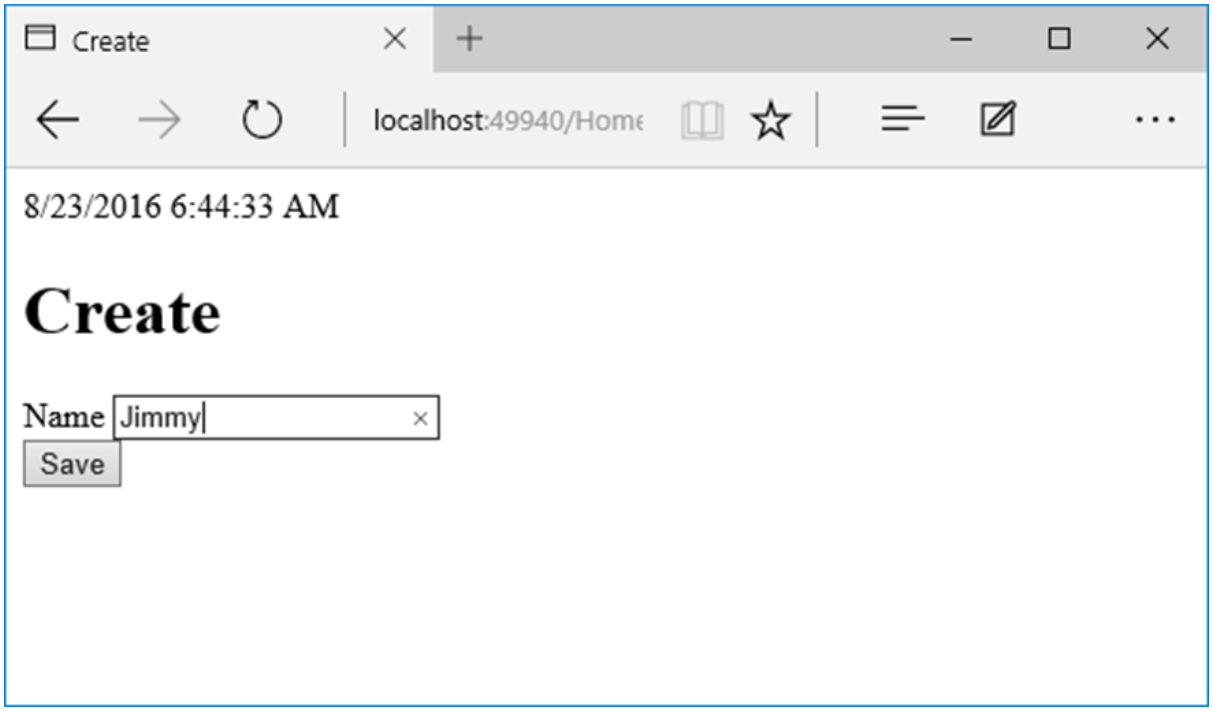
Let us add a link to **Create View** in the Index.cshtml file as shown in the following program.

```
@model HomePageViewModel

@{
    ViewBag.Title = "Home";
}
<h1>Welcome!</h1>
<table>
    @foreach (var employee in Model.Employees)
    {
        <tr>
            <td>@employee.Name</td>
            <td>
                <a asp-controller="Home" asp-action="Details" asp-route-
id="@employee.Id">Details</a>
                <a asp-controller="Home" asp-action="Edit" asp-route-
id="@employee.Id">Edit</a>
            </td>
        </tr>
    }
</table>
<div>
    <a asp-action="Create">Create</a>
</div>
```

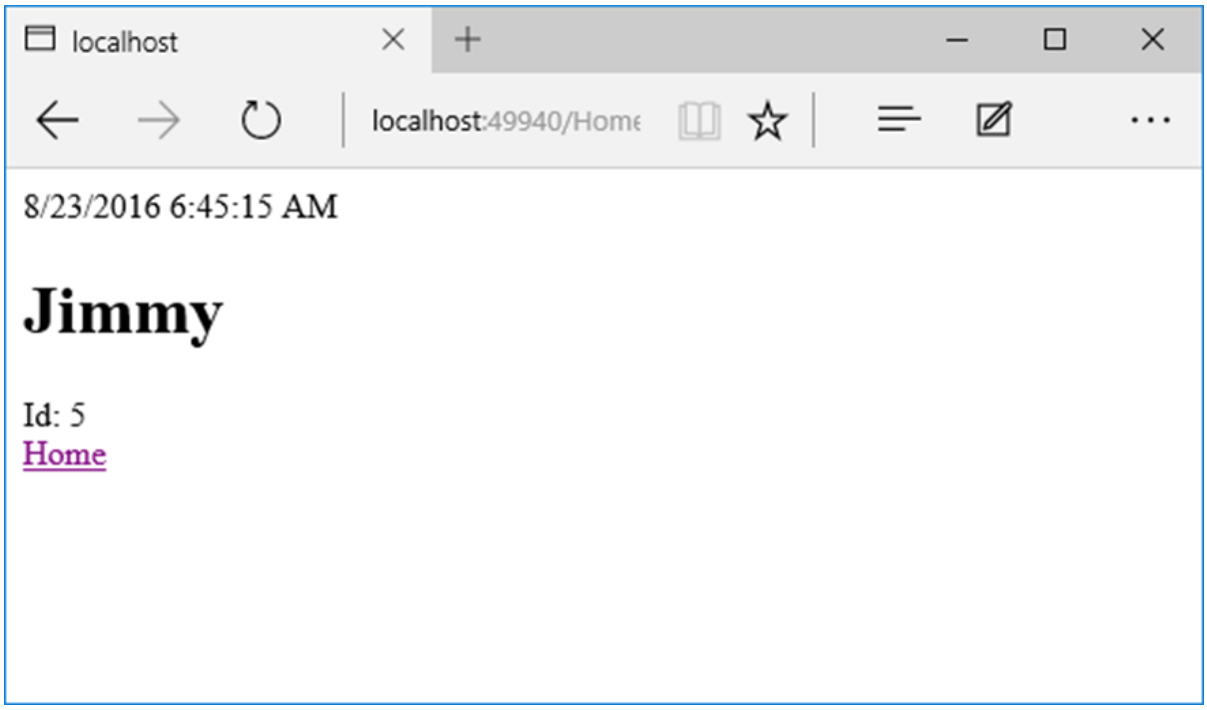
Run the application; you will see the following page.



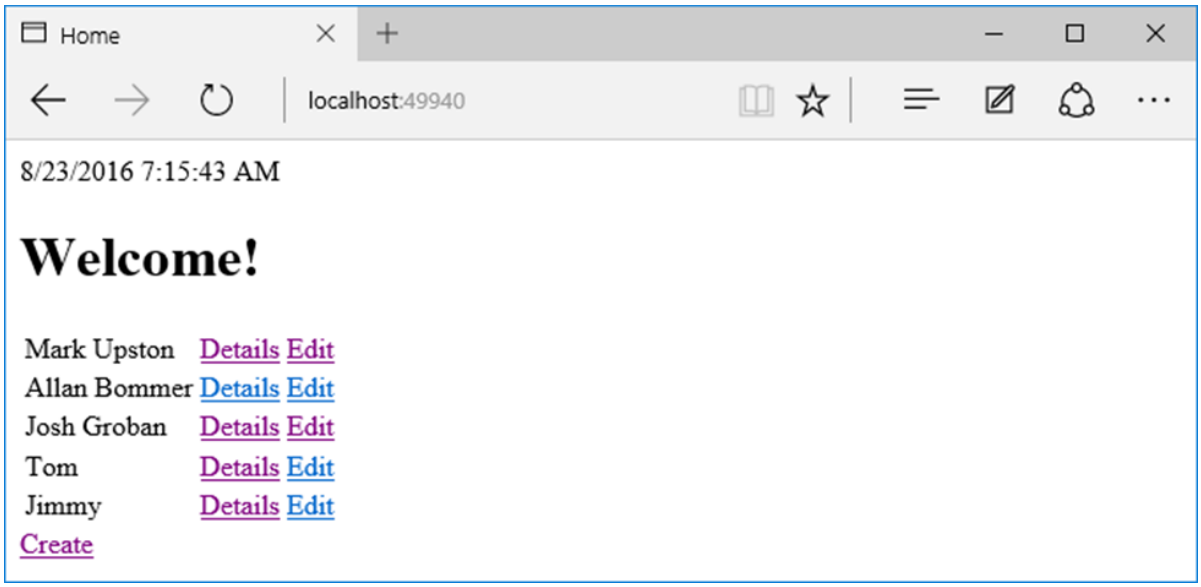
On the home page, you will see the Create link. When you click the Create link, it will take you to the Create View.



Enter a name in the Name field and click the Save button.



You will now see the **detail view** of the newly added employee. Let us click the Home link.



In this application, every user can create, edit an employee, and everyone can see the detail view. We want to change this behavior so that in the future, anonymous users can only see the list of employees on the home page, but every other action requires the user to identify themselves and sign in. We can do this with the **Authorize attribute**.

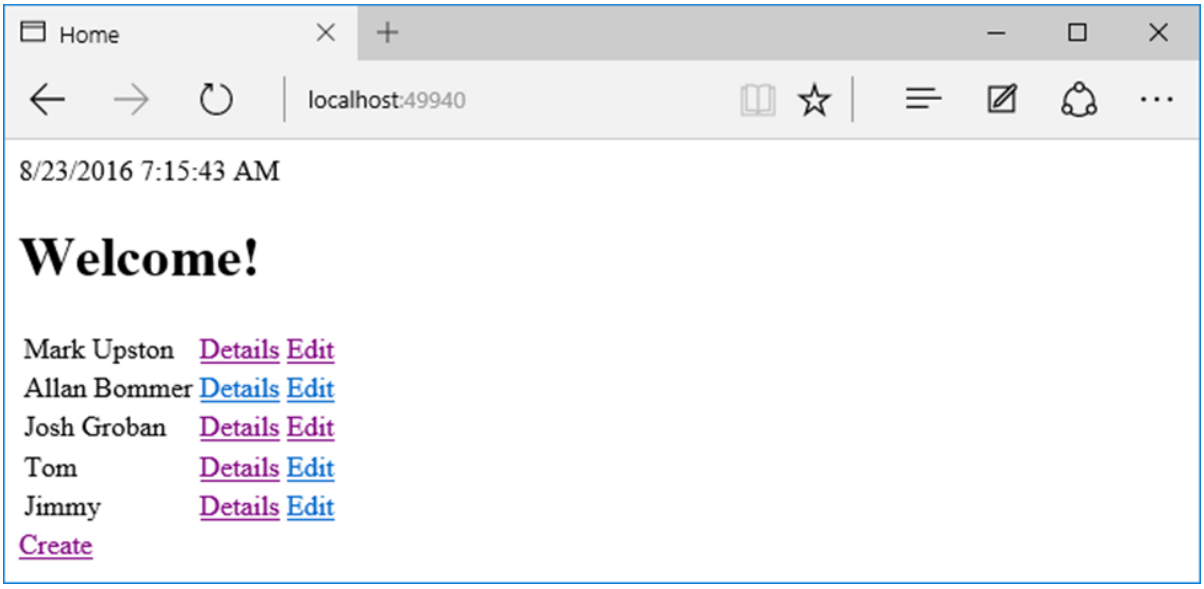
You can place the Authorize attribute on a controller or on individual actions inside the controller.

```
[Authorize]
public class HomeController : Controller
{
    //....
}
```

- When we place the Authorize attribute on the controller itself, the authorize attribute applies to all of the actions inside.
- The MVC framework will not allow a request to reach an action protected by this attribute unless the user passes an authorization check.
- By default, if you use no other parameters, the only check the Authorize attribute will make is a check to ensure the user is logged in so we know their identity.
- But you can use parameters to specify any fancy custom authorization policy that you like.
- There is also an **AllowAnonymous** attribute. This attribute is useful when you want to use the Authorize attribute on a controller to protect all of the actions inside, but then there is this single action or one or two actions that you want to unprotect and allow anonymous users to reach that specific action.

```
[AllowAnonymous]
 public ViewResult Index()
 {
     var model = new HomePageViewModel();
     using (var context = new FirstAppDemoDbContext())
     {
         SQLEmployeeData sqlData = new SQLEmployeeData(context);
         model.Employees = sqlData.GetAll();
     }

     return View(model);
 }
```

Let us try these attributes in our application. In the running application, an anonymous user can edit an employee.



We want to change this and force the users to log in and identify themselves before they can edit an employee. Let us now go into the HomeController. We will restrict access to one or two actions here. We can always place the Authorize attribute on those specific actions that we want to protect. We can also place the Authorize attribute on the controller itself, and this Authorize attribute is in the Microsoft.AspNet.Authorization namespace.

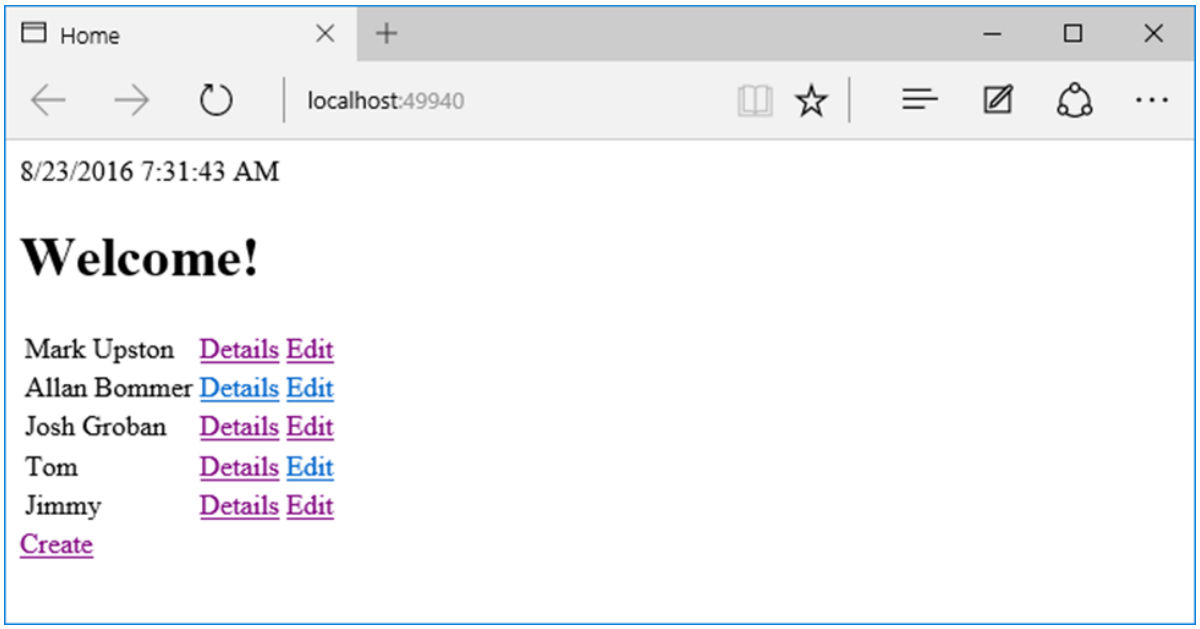
We will now use the Authorize attribute and force users to identify themselves to get into this controller except for the home page as shown in the following program.

```
[Authorize]
public class HomeController : Controller
{
    [AllowAnonymous]
    public ViewResult Index()
    {
        var model = new HomePageViewModel();
        using (var context = new FirstAppDemoDbContext())
        {
            SQLEmployeeData sqlData = new SQLEmployeeData(context);
            model.Employees = sqlData.GetAll();
        }

        return View(model);
    }

    public IActionResult Details(int id)
```

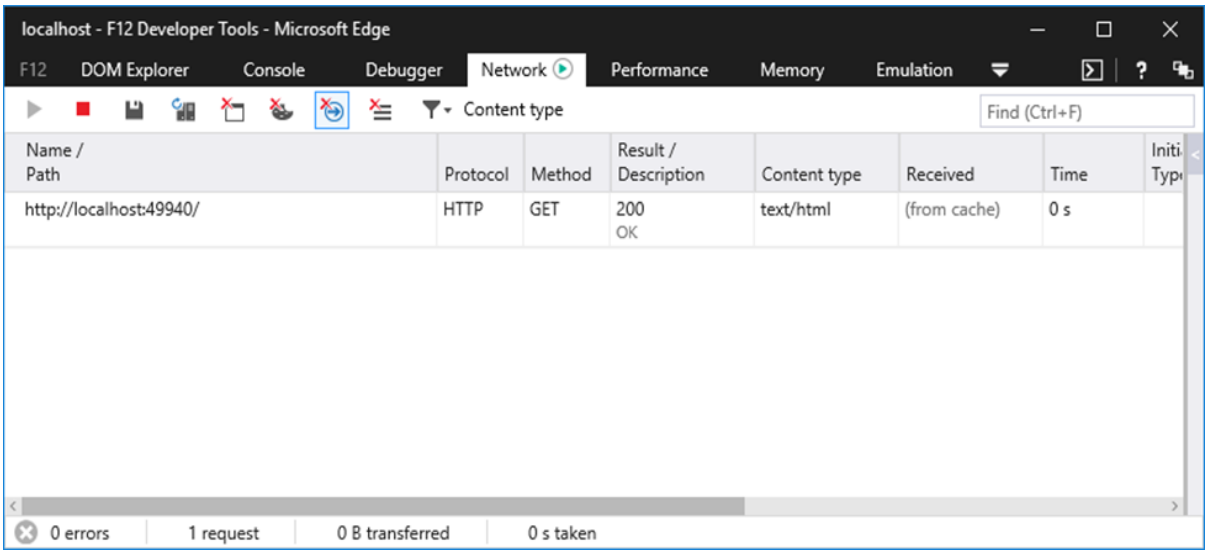
```
    {

        var context = new FirstAppDemoDbContext();
        SQLEmployeeData sqlData = new SQLEmployeeData(context);
        var model = sqlData.Get(id);
        if (model == null)
        {
            return RedirectToAction("Index");
        }
        return View(model);
    }
    [HttpGet]
    public IActionResult Edit(int id)
    {
        var context = new FirstAppDemoDbContext();
        SQLEmployeeData sqlData = new SQLEmployeeData(context);
        var model = sqlData.Get(id);
        if (model == null)
        {
            return RedirectToAction("Index");
        }
        return View(model);
    }

    [HttpPost]
    public IActionResult Edit(int id, EmployeeEditViewModel input)
    {
        var context = new FirstAppDemoDbContext();
        SQLEmployeeData sqlData = new SQLEmployeeData(context);
        var employee = sqlData.Get(id);
        if (employee != null && ModelState.IsValid)
        {
            employee.Name = input.Name;
            context.SaveChanges();

            return RedirectToAction("Details", new { id = employee.Id });
        }
        return View(employee);
```

```
    }

    [HttpGet]
    public ViewResult Create()
    {
        return View();
    }

    [HttpPost]
    public IActionResult Create(EmployeeEditViewModel model)
    {
        if (ModelState.IsValid)
        {
            var employee = new Employee();
            employee.Name = model.Name;

            var context = new FirstAppDemoDbContext();
            SQLEmployeeData sqlData = new SQLEmployeeData(context);
            sqlData.Add(employee);

            return RedirectToAction("Details", new { id = employee.Id });
        }
        return View();
    }
}
```

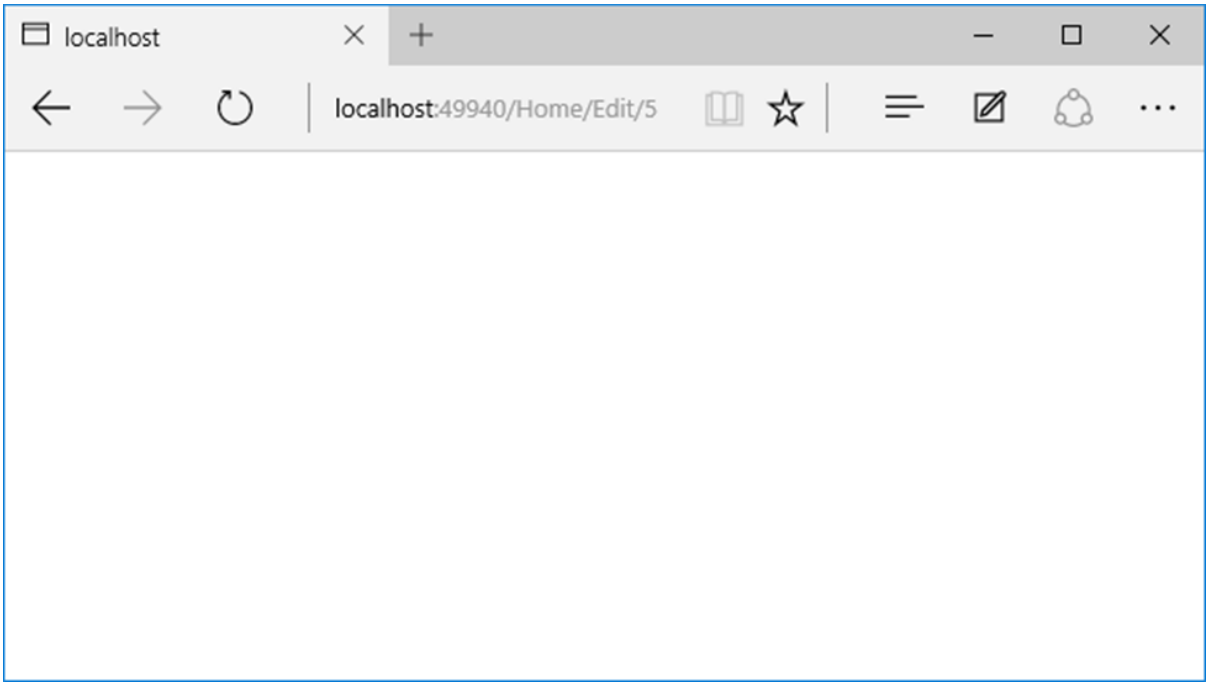
The home page or the **Index.cshtml** file that displays the list of employees have the **AllowAnonymous attribute**. Let us now run your application.



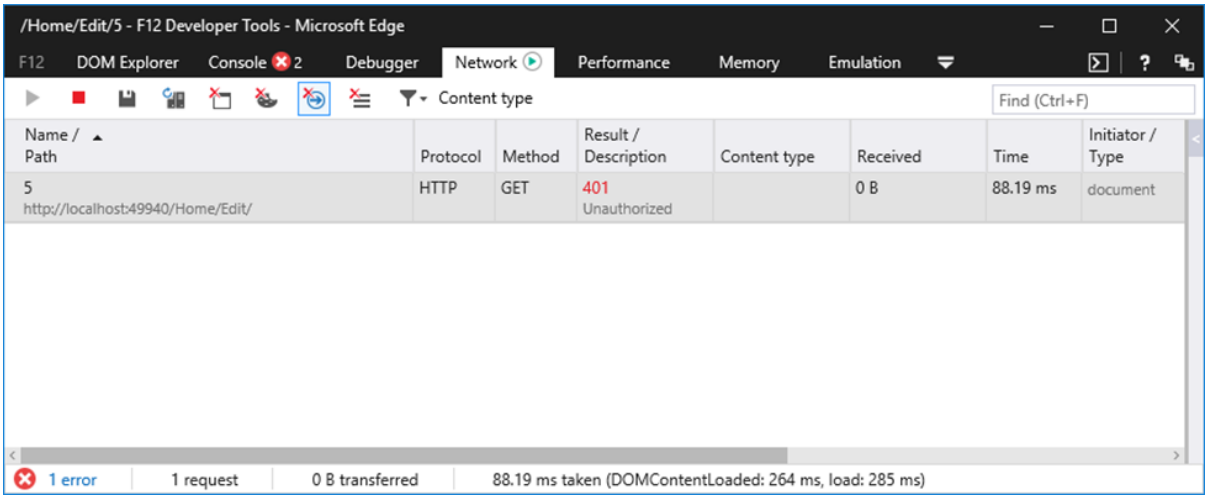
Press the F12 key which will open **developer tools**. Now, go to the **Network tab**.



There are a couple of things that we want to watch in the developer tools, so we can see how things work. When you click the Edit link, you will see a blank page.

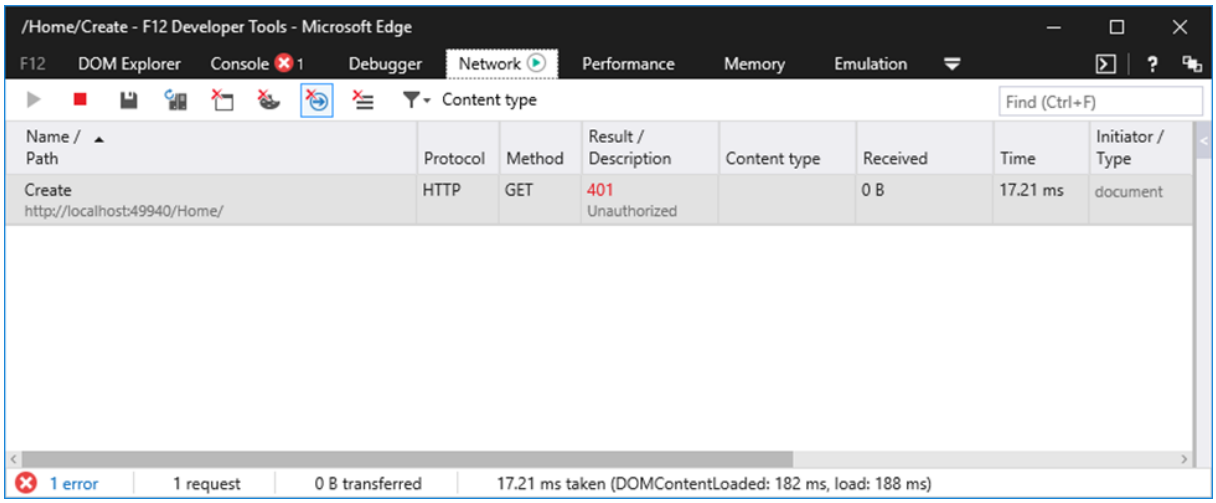


If you look at the developer tools, you will see that the HTTP status code that was returned from the server was a **401 status code**.



The 401 status code tells the browser that the request was not allowed through because it lacked valid authentication credentials. This tells us that the Authorize attribute is working.

Similarly, when you click the Create link on the home page, you will see the same error as shown in the following screenshot.



- Here, the bad part is that the user is left on a blank page and unless they have the developer tools open, they may not know that this was an authentication problem.

- This is where the Identity framework can step in and help.

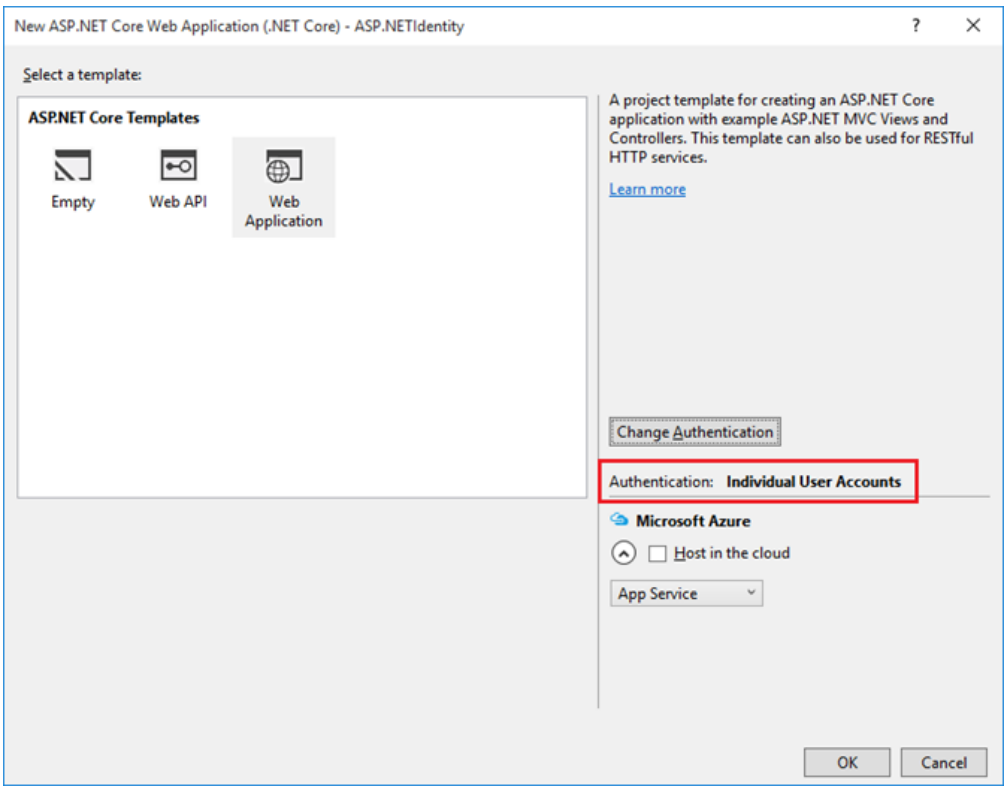
- The Identity framework can detect when a piece of the application wants to return the 401 status code because the user is not allowed to get there, and the Identity framework can turn that into a login page and allow the user to get past this problem.

- We will see how that works once we get the Identity framework installed and configured.

- But right now, we can see that the **Authorize attribute** is working.

25. ASP.NET Core — Identity Configuration

In this chapter, we will install and configure the Identity framework, which takes just a little bit of work. If you go to the Visual Studio and create a new ASP.NET Core application, and you select the full web application template with authentication set to individual user accounts, that new project will include all the bits of the Identity framework set up for you.

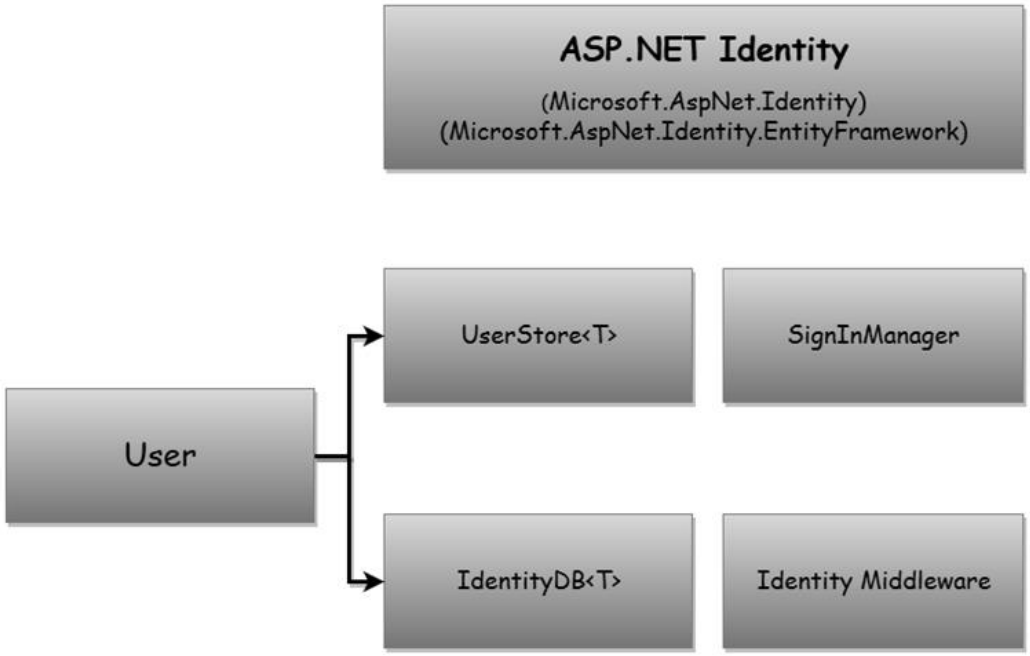


We started from an empty project. We will now set up the Identity framework from scratch, which is a good way to learn about all the pieces that are in the full application template because it can be confusing if you haven't worked your way through all the codes in detail.

To get started, we will need to install the dependency, which is **Microsoft.AspNet.Identity**. We will proceed by installing **Microsoft.AspNet.Identity.EntityFramework** and then, implement the Identity framework that works with the Entity Framework.

- If we take a dependency on Identity.EntityFramework, the package is inclusive of the Identity package.
- If you build your own data stores, you can work just with the Identity package.
- Once our dependencies are installed, we can create a customer User class with all the information we want to store about a user.

- For this application, we are going to inherit from a class provided by the Identity framework and that class will give us all the essentials like the Username property and a place to store the hashed passwords.



- We will also need to modify our **FirstAppDemoDbContext** class to inherit from the Identity framework's **IdentityDb** class.

- The IdentityDb gives us everything we need to store as user information with the Entity Framework. Once we have a User class and a **DBContext** set up, we will need to configure the Identity services into the application with the **ConfigureServices** method of the Startup class.

- Just like when we needed to add services to support the MVC framework, the Identity framework needs services added to the application in order to work.

- These services include services like the **UserStore** service and the **SignInManager**.

- We will be injecting those services into our controller to create users and issue cookies at the appropriate time.

- Finally, during the Configure method of startup, we will need to add the Identity middleware.

- This middleware will not only help to turn cookies into a user identity, but also make sure that the user doesn't see an empty page with a 401 response.

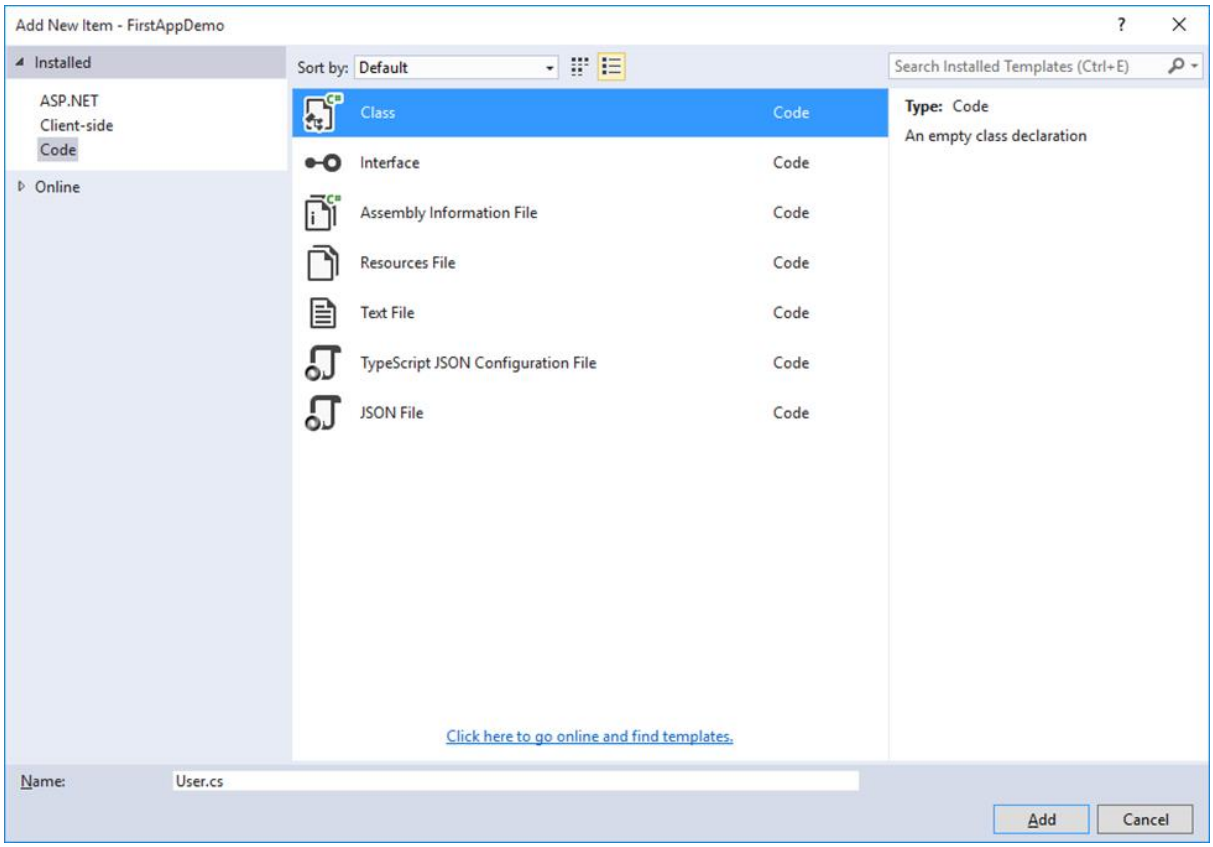
Let us now follow the steps given below.

Step 1: We need to proceed by adding a dependency on the Identity framework. Let us add Microsoft.AspNet.Identity.EntityFramework dependency into the project.json file. This will include all of the other necessary Identity packages that we need.

```
{
  "version": "1.0.0-*",
  "compilationOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Diagnostics": "1.0.0-rc1-final",
    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "EntityFramework.Commands": "7.0.0-rc1-final",
    "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-rc1-final",
    "Microsoft.AspNet.Identity.EntityFramework": "3.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel",
    "ef": "EntityFramework.Commands"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
```

```
  "publishExclude": [
     "**.user",
     "**.vspscc"
   ]
}
```

Step 2: Save this file. The Visual Studio restores the packages and now, we can add our User class. Let us add the User class by right-clicking on the Models folder and selecting **Add > Class**.



Call this class User and click on the Add button as in the above screenshot. In this class, you can add properties to hold any information that you want to store about a user.

Step 3: Let us derive the User class from a class provided by the Identity framework. It is the IdentityUser class that is in the Identity.EntityFramework namespace.

```
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Models
{
```

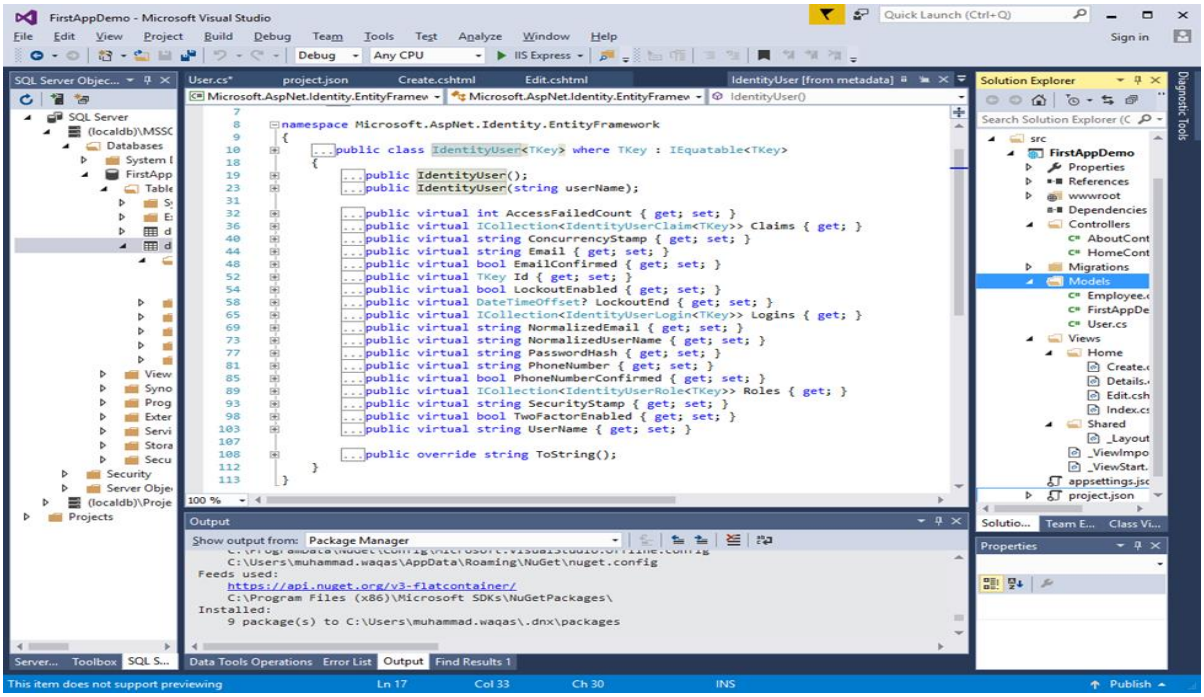
```
    public class User : IdentityUser
    {
    }
}
```

Step 4: Let us now go to the IdentityUser, put the cursor on that symbol, and press F12 to see the Visual Studio's metadata view.

```
#region Assembly Microsoft.AspNet.Identity.EntityFramework, Version=3.0.0.0,

namespace Microsoft.AspNet.Identity.EntityFramework
{
    public class IdentityUser : IdentityUser<string>
    {
        public IdentityUser();
        public IdentityUser(string userName);
    }
}
```

Step 5: You can see that IdentityUser is derived from the IdentityUser of the string. You can change the type of the primary key by deriving from the IdentityUser and specifying our generic type parameter. You can also store things with a primary key that is ideally an integer value.

Step 6: Let us now place the cursor on the IdentityUser of string and press F12 again to go to the metadata view.



You can now see all the information related to a user by default. The information includes the following:

- The fields that we won't use in this application, but are available for use.
- The Identity framework can keep track of the number of failed login attempts for a particular user and can lock that account over a period of time.
- The fields to store the PasswordHash, the PhoneNumber. The two important fields that we will be using are the PasswordHash and the UserName.
- We will also be implicitly using the primary key and the ID property of a user. You can also use that property if you need to query for a specific user.

Step 7: Now, we need to make sure that the User is included in our DBContext. So, let us open the **FirstAppDemoDBContext** that we have in our application, and instead of deriving it directly from the DBContext, which is the built-in Entity Framework base class, we now need to derive it from the IdentityDbContext.

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Data.Entity;


namespace FirstAppDemo.Models
{
    public class FirstAppDemoDbContext : IdentityDbContext<User>
    {
        public DbSet<Employee> Employees { get; set; }


        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Data
Source=(localdb)\\MSSQLLocalDB;Initial Catalog=FirstAppDemo;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrit
e;MultiSubnetFailover=False");
        }
    }
}
```

Step 8: The IdentityDbContext class is also in the Microsoft.AspNet.Identity.EntityFramework namespace and we can specify the type of user it should store. This way, any additional fields we add to the User class gets into the database.

- The IdentityDbContext brings additional DbSets, not just to store a user but also information about the user roles and the user claims.
- Our User class is ready now. Our FirstAppDemoDbContext class is configured to work with the Identity framework.

- We can now go into Configure and ConfigureServices to set up the Identity framework.

Step 9: Let us now start with **ConfigureServices**. In addition to our MVC services and our Entity Framework services, we need to add our Identity services. This will add all the services that the Identity framework relies on to do its work.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<FirstAppDemoDbContext>(option =>
option.UseSqlServer(Configuration["database:connection"]));

    services.AddIdentity<User, IdentityRole>()
        .AddEntityFrameworkStores<FirstAppDemoDbContext>();
}
```

- The AddIdentity method takes two generic type parameters — the type of user entity and the type of role entity.

- The two generic type parameters are the types of our user — the User class we just created and the Role class that we want to work with. We will now use the built-in IdentityRole. This class is in the EntityFramework namespace.

- When we are using the Entity Framework with Identity, we also need to invoke a second method — the AddEntityFrameworkStores.

- The AddEntityFrameworkStores method will configure services like the UserStore, the service used to create users and validate their passwords.

Step 10: The following two lines are all we need to configure the services for the application.

```
services.AddIdentity<User, IdentityRole>()
        .AddEntityFrameworkStores<FirstAppDemoDbContext>();
```

Step 11: We also need to add the middleware. The location of where we insert the middleware is important because if we insert the middleware too late in the pipeline, it will never have the chance to process a request.

And if we require authorization checks inside our MVC controllers, we need to have the Identity middleware inserted before the MVC framework to make sure that cookies and also the 401 errors are processed successfully.

```
public void Configure(IApplicationBuilder app)
{
```

```
    app.UseIISPlatformHandler();

    app.UseDeveloperExceptionPage();
    app.UseRuntimeInfoPage();

    app.UseFileServer();

    app.UseIdentity();
    app.UseMvc(ConfigureRoute);

    app.Run(async (context) =>
    {
        var msg = Configuration["message"];
        await context.Response.WriteAsync(msg);
    });

}
```

Step 12: The location where we insert the middleware is where we will add the Identity middleware. The following is the complete implementation of the Startup.cs file.

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using FirstAppDemo.Services;
using Microsoft.AspNet.Routing;
using System;
using FirstAppDemo.Entities;
using Microsoft.Data.Entity;
using FirstAppDemo.Models;
using Microsoft.AspNet.Identity.EntityFramework;

namespace FirstAppDemo
{
    public class Startup
    {
        public Startup()
        {
```

```
            var builder = new ConfigurationBuilder()
                            .AddJsonFile("AppSettings.json");
            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
            services.AddEntityFramework()
                .AddSqlServer()
                .AddDbContext<FirstAppDemoDbContext>(option =>
option.UseSqlServer(Configuration["database:connection"]));

            services.AddIdentity<User, IdentityRole>()
                .AddEntityFrameworkStores<FirstAppDemoDbContext>();
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.UseIISPlatformHandler();

            app.UseDeveloperExceptionPage();
            app.UseRuntimeInfoPage();

            app.UseFileServer();

            app.UseIdentity();
            app.UseMvc(ConfigureRoute);

            app.Run(async (context) =>
            {
```

```
                var msg = Configuration["message"];
                await context.Response.WriteAsync(msg);
            });
        }

        private void ConfigureRoute(IRouteBuilder routeBuilder)
        {
            //Home/Index
            routeBuilder.MapRoute("Default",
"{controller=Home}/{action=Index}/{id?}");
        }

        // Entry point for the application.
        public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
    }
}
```

Step 13: Let us now move ahead by building the application. In the next chapter, we need to add another Entity Framework migration to make sure we have the Identity schema in our SQL Server database.

26. ASP.NET Core — Identity Migrations

In this chapter, we will discuss the Identity migration. In ASP.NET Core MVC, authentication and identity features are configured in the Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<FirstAppDemoDbContext>(option =>
option.UseSqlServer(Configuration["database:connection"]));

    services.AddIdentity<User, IdentityRole>()
        .AddEntityFrameworkStores<FirstAppDemoDbContext>();
}
```

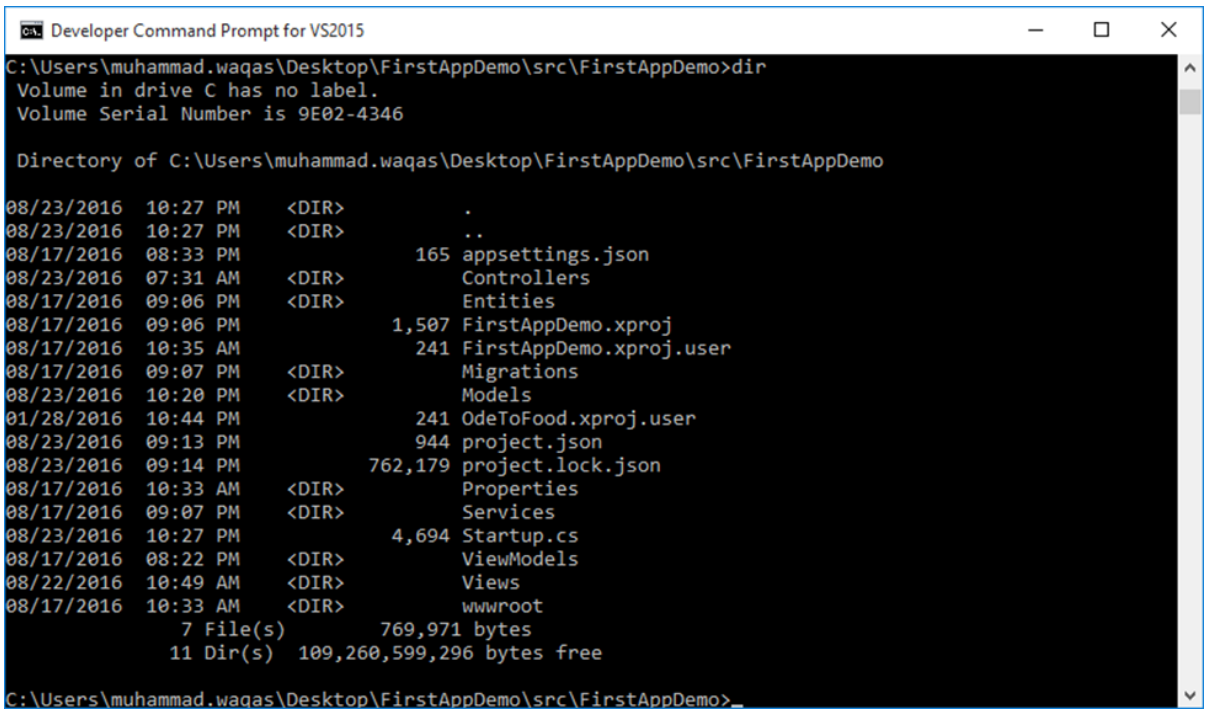
Anytime you make a change to one of your entity classes or you make a change to your DBContext derived class, chances are you will have to create a new migration script to apply to the database and bring the schema in sync with what is in your code.

This is the case in our application because we now derive our FirstAppDemoDbContext class from the IdentityDbContext class, and it contains its own DbSets, and it will also create a schema to store all the information about the entities that it manages.

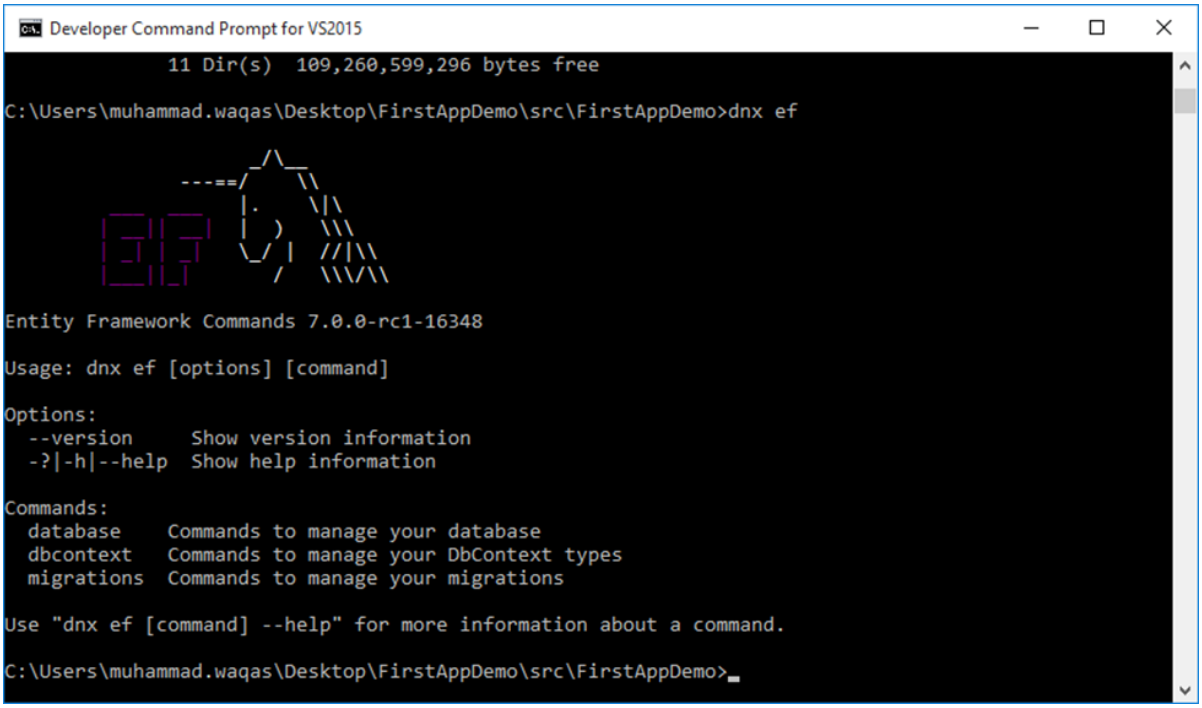
```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Data.Entity;

namespace FirstAppDemo.Models
{
    public class FirstAppDemoDbContext : IdentityDbContext<User>
    {
        public DbSet<Employee> Employees { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Data
Source=(localdb)\\MSSQLLocalDB;Initial Catalog=FirstAppDemo;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrit
e;MultiSubnetFailover=False");
        }
```

```
    }
}
```

Let us now open the command prompt and make sure we are in the location where the project.json file exists for our project.



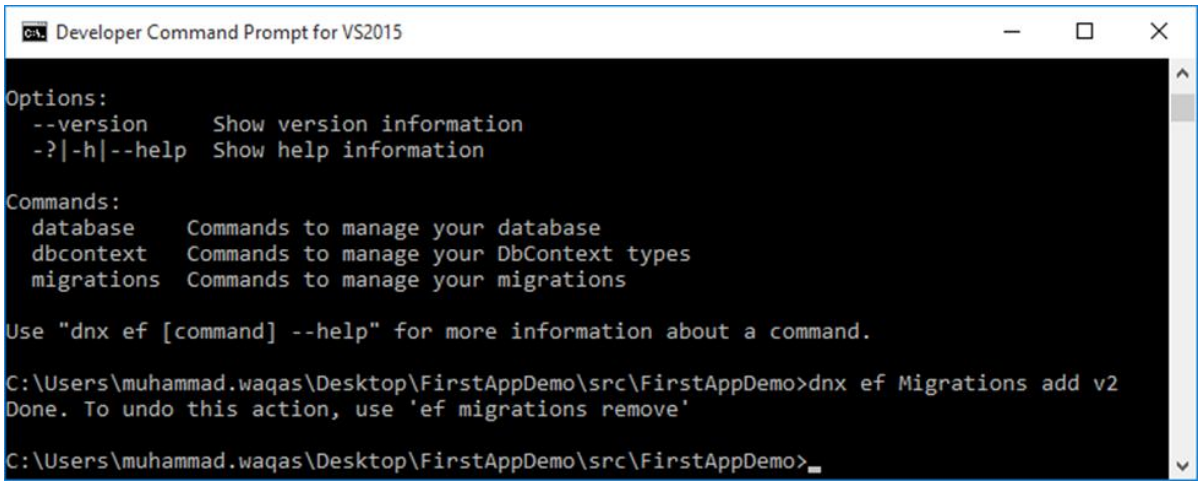
We can also get the Entity Framework commands by typing **dnx ef**.



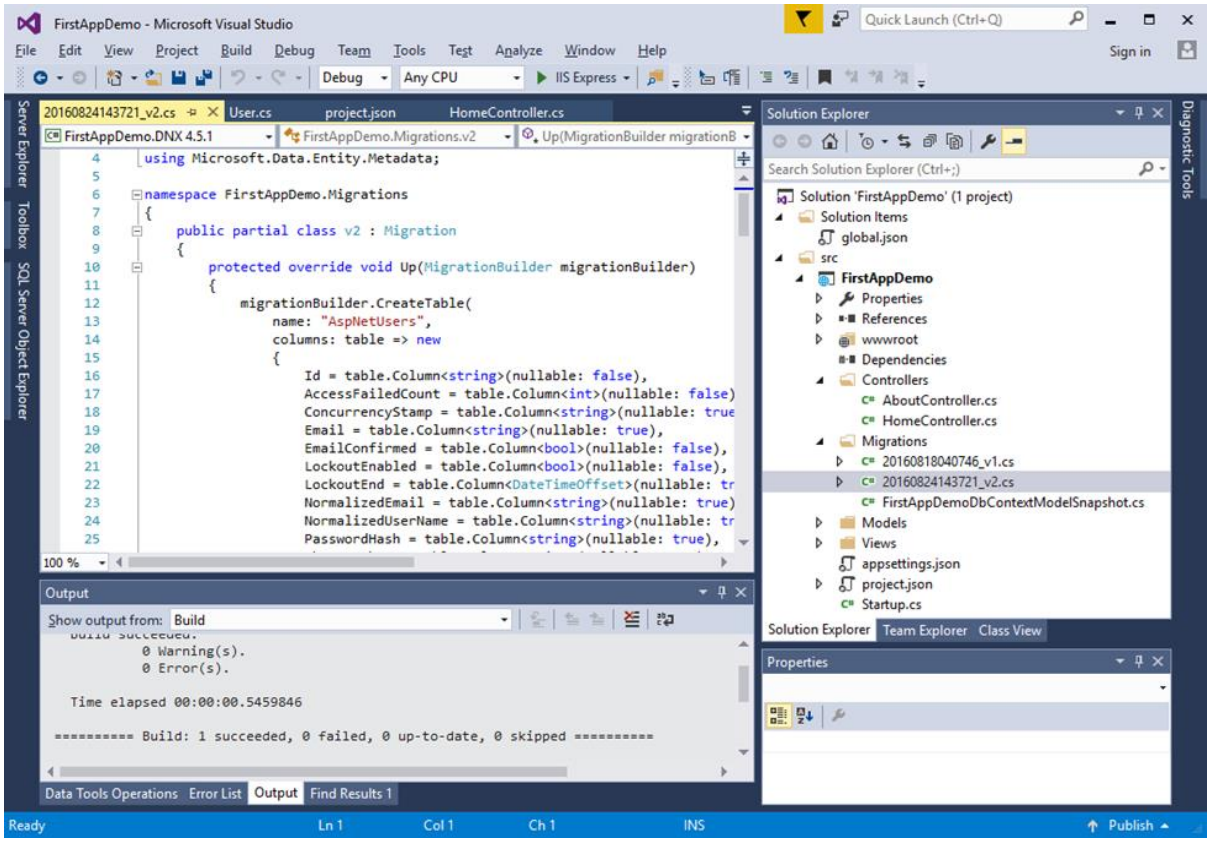
Our project.json file has a section that maps this "**ef**" keyword with the EntityFramework.Commands.

```
"commands": {
    "web": "Microsoft.AspNet.Server.Kestrel",
    "ef": "EntityFramework.Commands"
  }
```

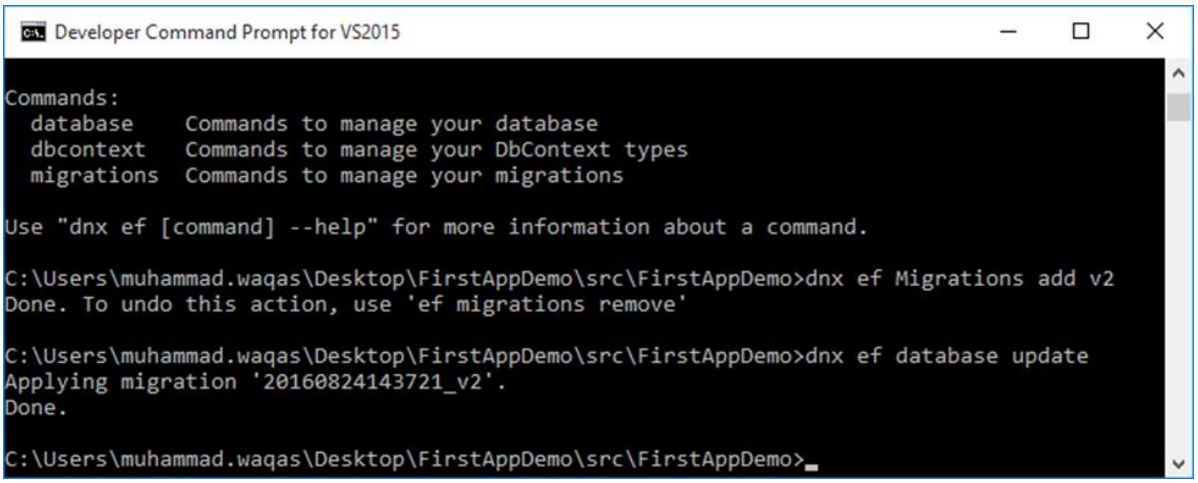
We can add a migration from here. We also need to provide a name to the migration. Let us use v2 for version 2 and press enter.



When the migration is complete, you will have a v2 file in your migrations folder.

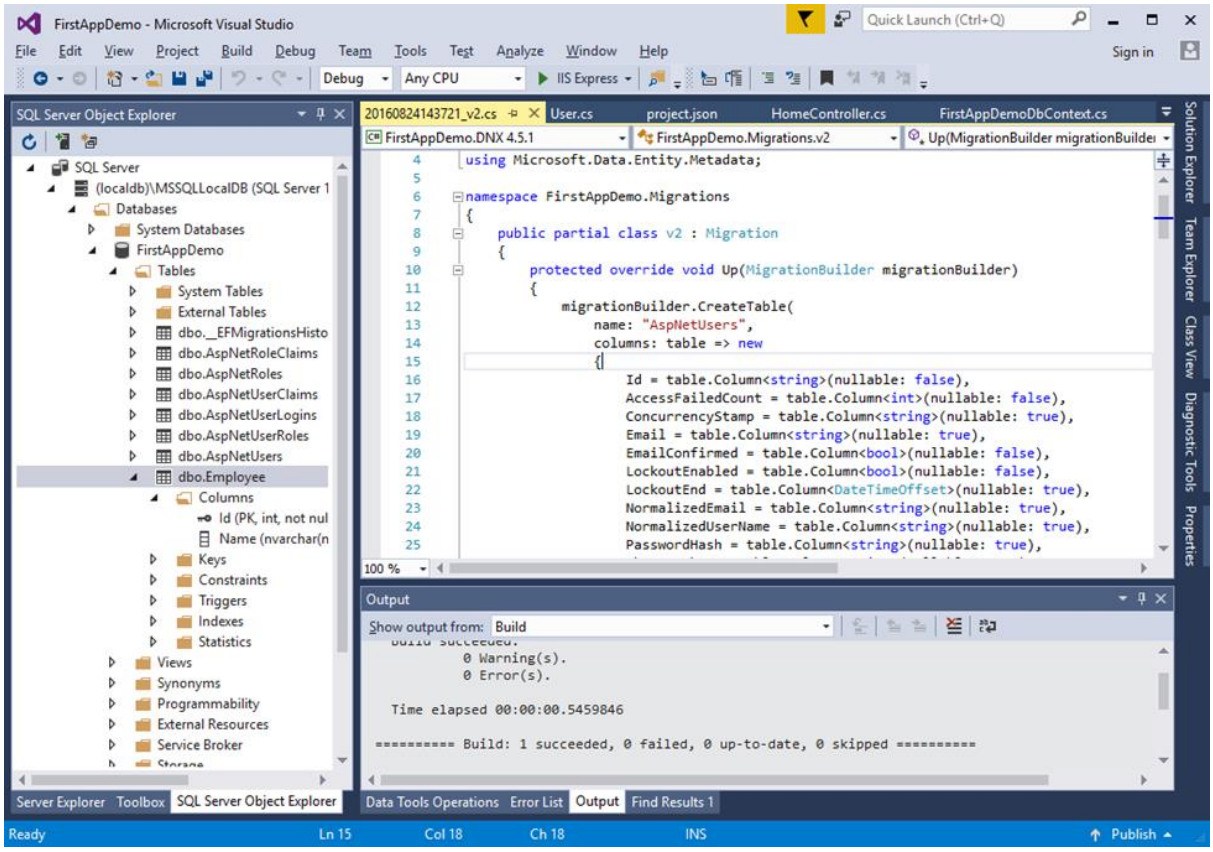


We now want to apply that migration to our database by running the "**dnx ef database update**" command.



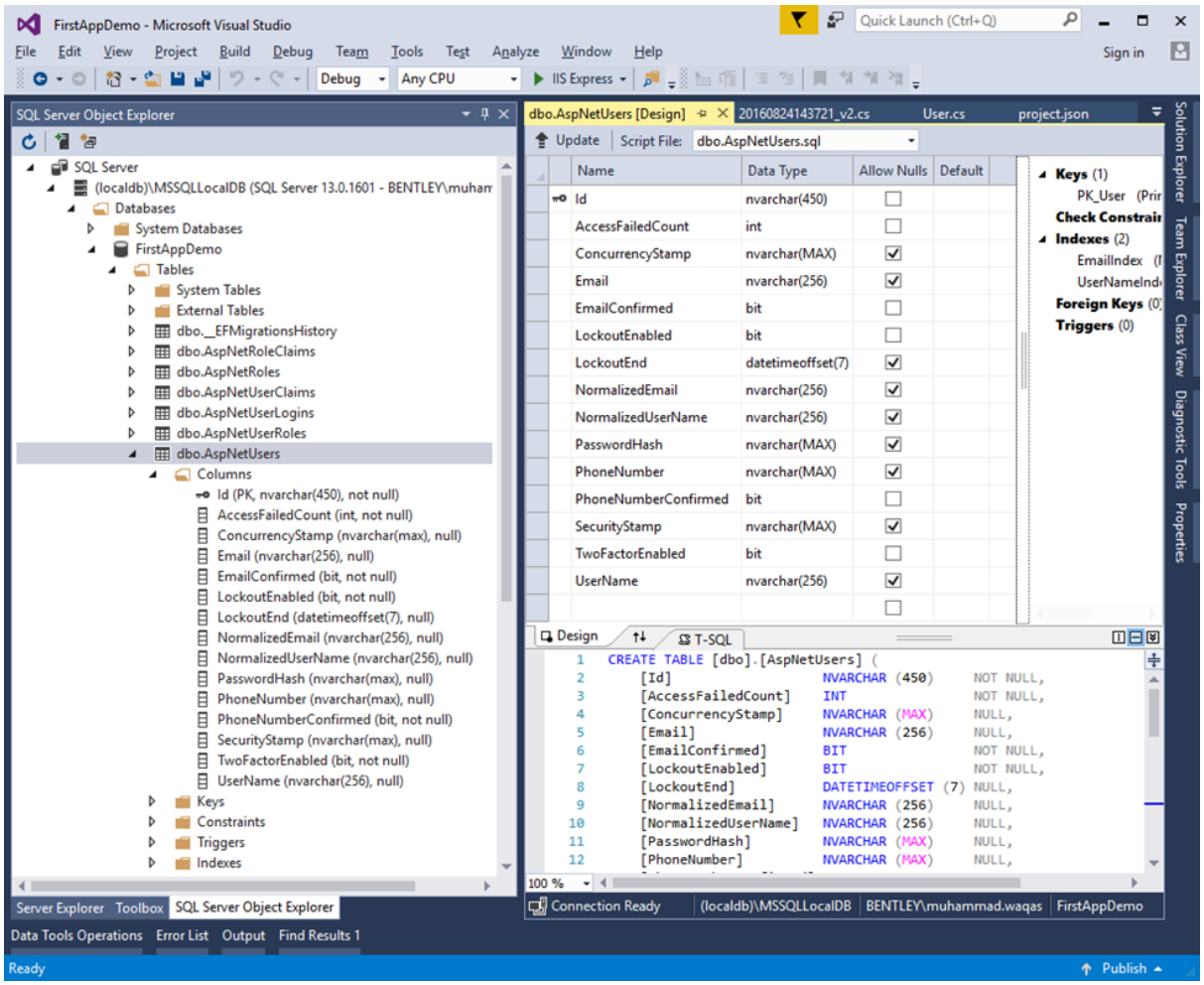
The Entity Framework will see there is a migration that needs to be applied and it will execute that migration.

If you come into the SQL Server Object Explorer, you will see the Employee table that we created earlier. You will also see some additional tables that have to store users, claims, roles, and some mapping tables that map users to specific roles.



All these tables are related to the entities that the Identity framework provides.

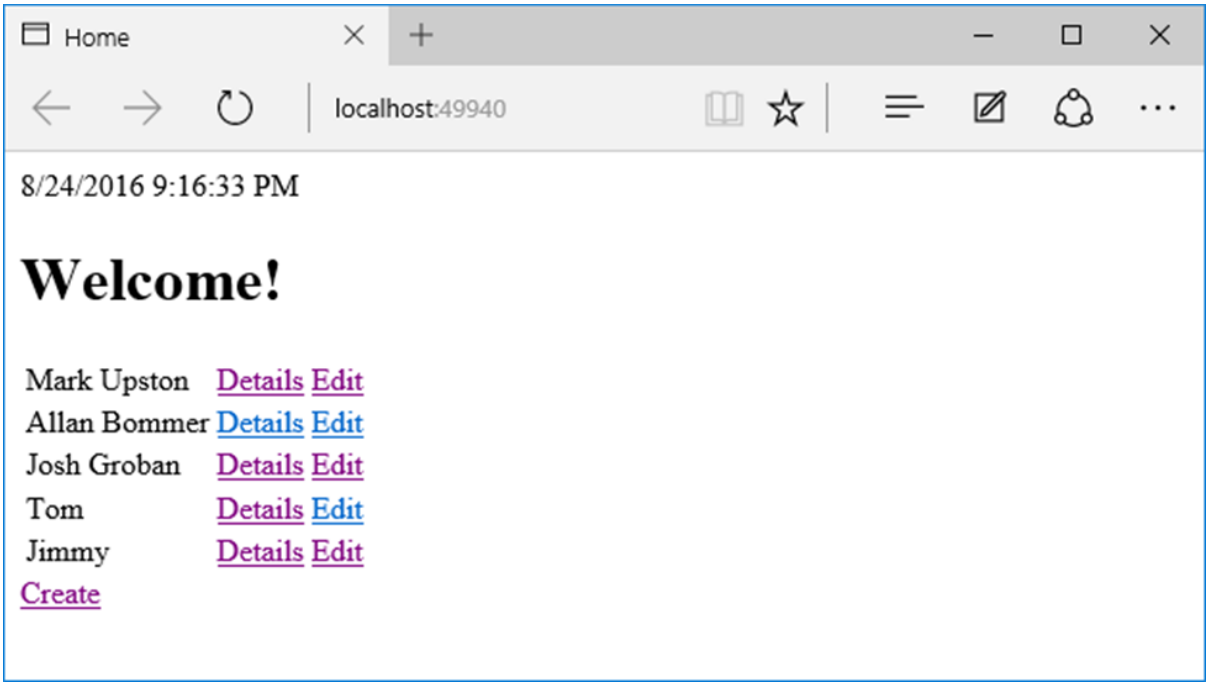
Let us take a quick look at the **users table**.



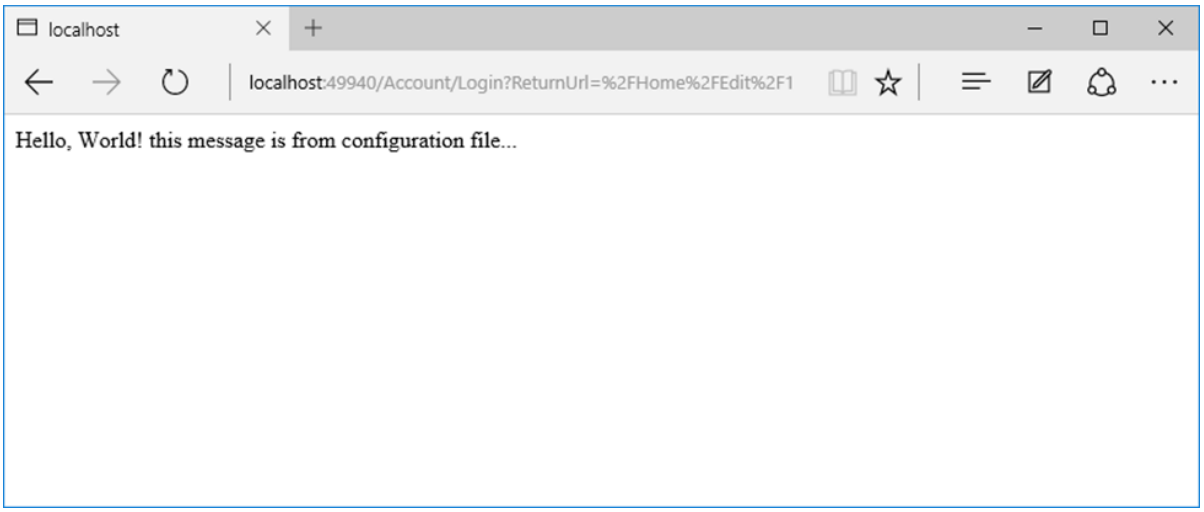
You can now see that the columns in the AspNetUsers table include columns to store all those properties that we saw on the Identity User which we inherited from, and its fields like UserName and PasswordHash. So, you have been using some of the built-in Identity services because they also contain an ability to create a user and validate a user's password.

27. ASP.NET Core — User Registration

In this chapter, we will discuss the User registration. We now have a working database, and it is time to start adding some of the features to the application. We have also configured our application and we have a working database schema. Let us now go to the application home page.

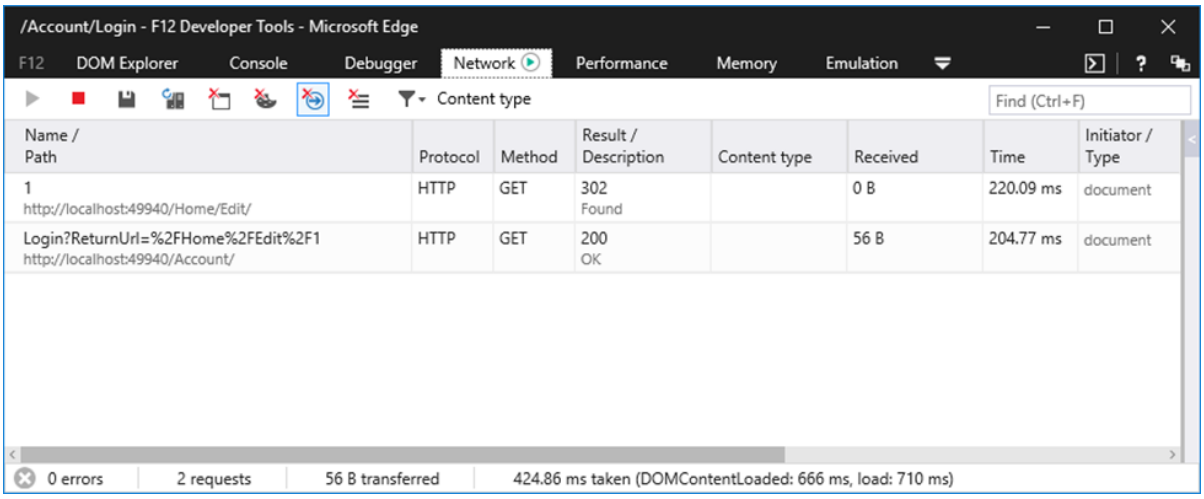


Open the developer tools by pressing F12 and then click on the Edit link. Previously, when we clicked on the Edit link, the MVC framework detected the presence of the Authorize attribute and returned a 401 status code because the user wasn't logged in.



You will now see that we get a message on the screen from the configuration file.

Let us now go to the developer tools.



- You will see that the browser requested for the edit page, and the MVC framework decided the user is not authorized to view this resource.
- So somewhere inside the MVC framework, a 401 status code was generated.
- We now have the Identity middleware in place. The Identity middleware looks at that 401 status code that's going to go out to the user and replaces it with a 302 status code, which is a redirect status code.
- The Identity framework knows that the user will have to try to log in before it can reach this resource.
- The Identity framework directed us to this URL, as we can see in the address bar — /Account/Login.
- This is a configurable endpoint with the Identity framework, inside the Startup when you are registering these services and the middleware. There are different options you can set and one of the options is to change the login URL.
- By default, the URL is going to be /Account/Login. Currently, we don't have an account controller, so ultimately what we want to do is to create an account controller and allow a user to log in.
- But before the users can even log in, they will need to register on the site and save their usernames and a passwords.
- Both the login and the register features can be part of an account controller.

Let us now move ahead and add a new class in Controllers folder, and call it the AccountController. We will derive this from the MVC framework base Controller class.

```
using Microsoft.AspNet.Mvc;
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
```

```
using System.Linq;
using System.Threading.Tasks;

namespace FirstAppDemo.Controllers
{
    public class AccountController : Controller
    {

    }
}
```

- We will now have to set up a feature where a user can register for this site.
- It is going to be very much like an edit form.
- When the user wants to register, we will first display a form that allows them to fill the required information. They can then upload this form to the site.
- This information is then saved on the database.

Let us now create the action that will return a view when we go to /account/register.

```
public class AccountController : Controller
{
    [HttpGet]
    public ViewResult Register()
    {
        return View();
    }
}
```

We don't need to look anything up, the user will provide all the information that we need. Before we build the ViewModel for that view, we need to decide on the information that the view will display. We also need to decide on the information that we will need to receive from the user?

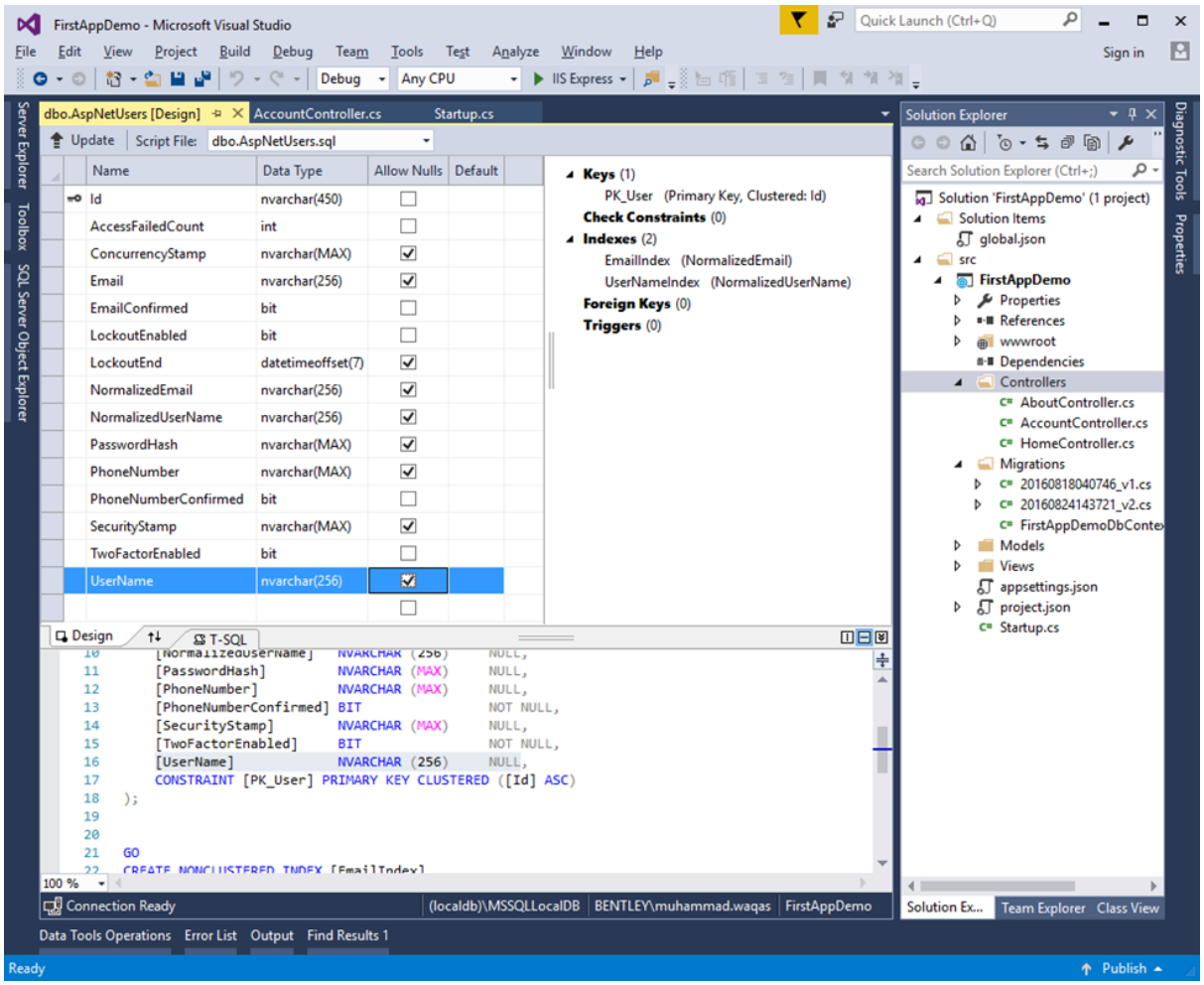
Let us create a view model for this scenario by adding a new class in the AccountController.cs file and call this the RegisterViewModel.

Let us create some properties that will hold the Username, Password and also the user ConfirmPassword by typing it twice and making sure that both passwords match as shown in the following program.

```
public class RegisterViewModel
```

```
{
    [Required, MaxLength(256)]
    public string Username { get; set; }

    [Required, DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password), Compare(nameof(Password))]
    public string ConfirmPassword { get; set; }
}
```

In the above class, you can see some annotations that can help us validate this model. The username is required here and if you look at the database schema, the column to hold a username is 256 characters long.

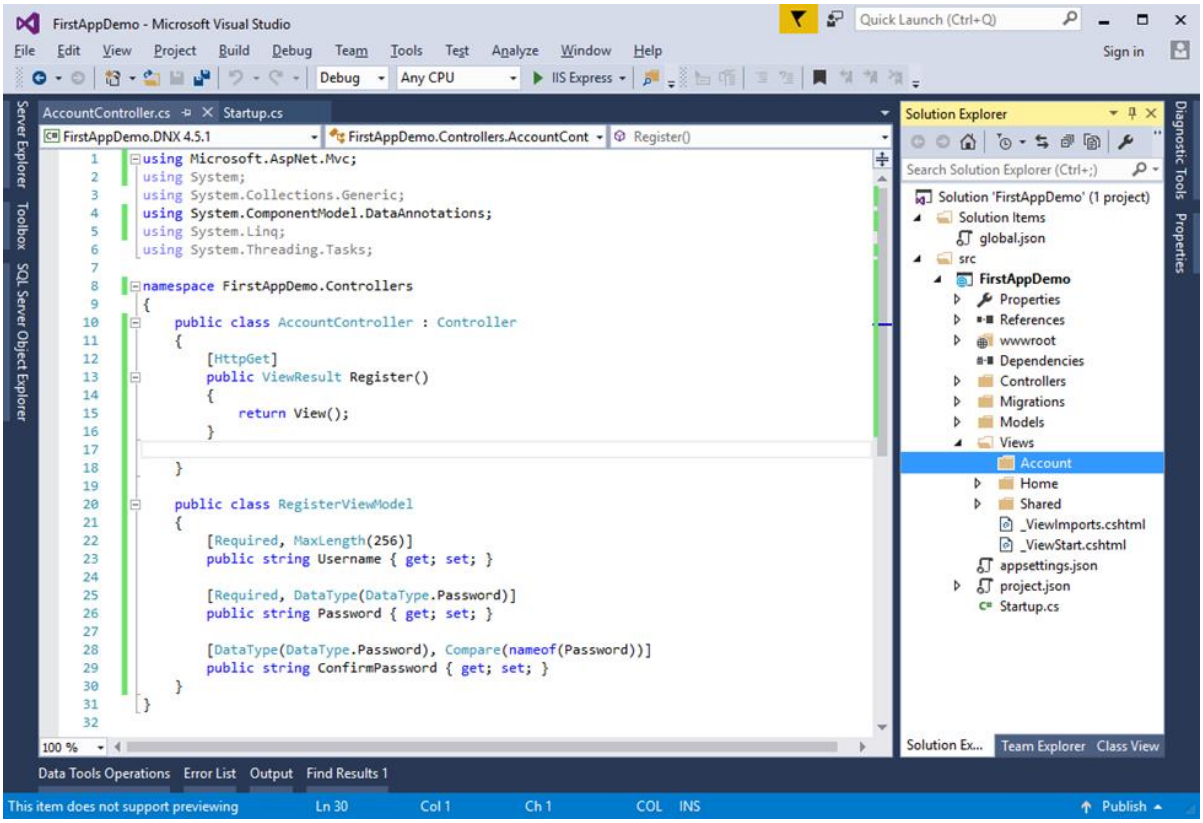


- We will also apply a MaxLength attribute here.

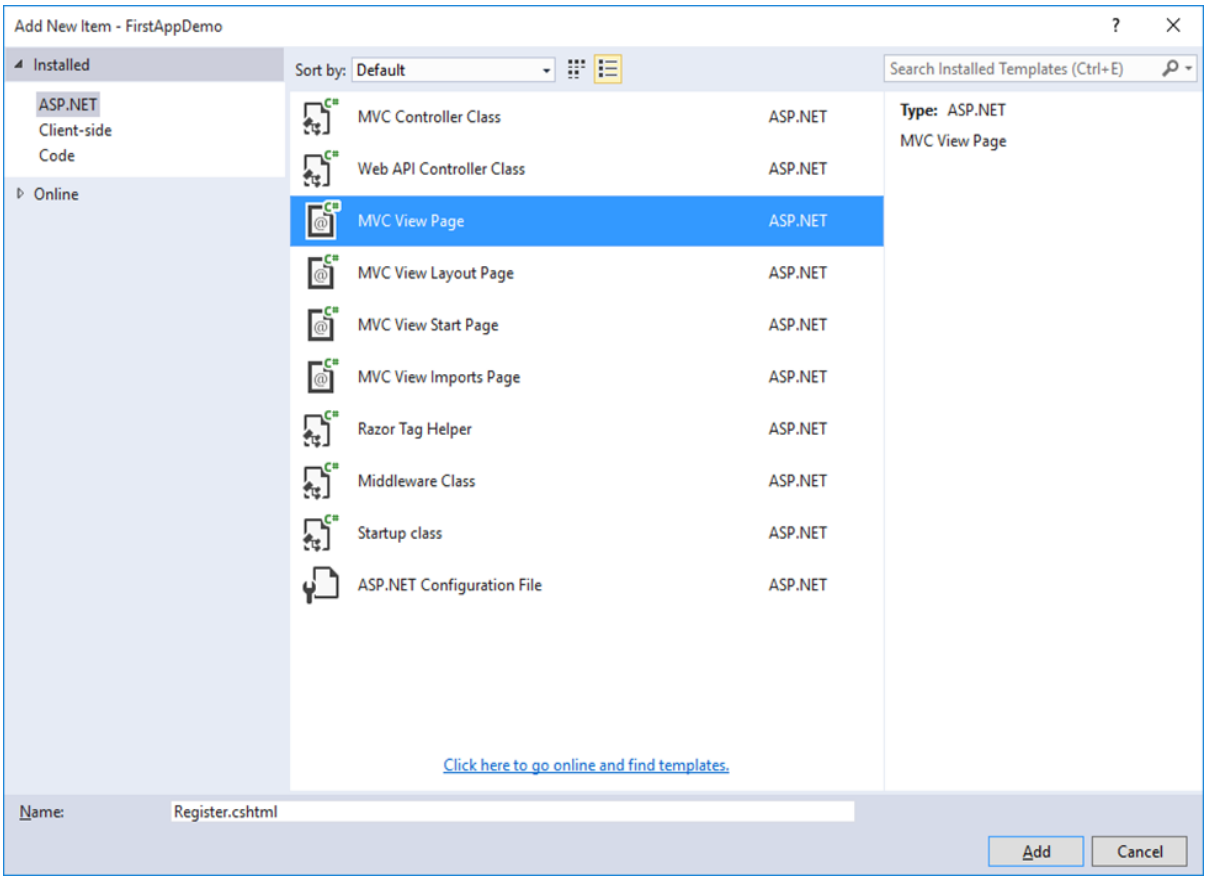
- A password is going to be required, and when we render an input for this password, we want the input type to be of **Type Password** so that the characters don't display on the screen.

- The **Confirm Password** is also going to be the DataType Password and then there is an additional Compare attribute. We will compare the ConfirmPassword field with this other property that we can specify, which is the Password field.

Let us now create the view that we need. We will need to add a new folder to the views and call it Account, so all of the views related to the AccountController will be added in this folder.



Now, right-click on the Account folder and select Add > New Item.



In the middle pane, select the MVC View Page and call it Register.cshtml and then Click on the Add button.

Delete all the existing codes from the Register.cshtml file and add the following code.

```
@model RegisterViewModel
@{
    ViewBag.Title = "Register";
}

<h1>Register</h1>
<form method="post" asp-controller="Account" asp-action="Register">
    <div asp-validation-summary="ValidationSummary.ModelOnly"></div>
    <div>
        <label asp-for="Username"></label>
        <input asp-for="Username" />
        <span asp-validation-for="Username"></span>
    </div>
    <div>
        <label asp-for="Password"></label>
```

```
        <input asp-for="Password" />
        <span asp-validation-for="Password"></span>
    </div>
    <div>
        <label asp-for="ConfirmPassword"></label>
        <input asp-for="ConfirmPassword" />
        <span asp-validation-for="ConfirmPassword"></span>
    </div>
    <div>
        <input type="submit" value="Register" />
    </div>
</form>
```

- You can now see that we have specified the model as the RegisterViewModel that we have just created.
- We will also set the title for this page using the ViewBag and we want the title to be Register.
- We also need to create a form which contains the fields for Username, Password and ConfirmPassword.
- We have also included a div that will display a validation summary. When we use the ASP validation summary, we need to specify what errors are to appear in the summary.
- We can have all the errors appear in the summary area, or we can say ValidationSummary.ModelOnly and the only errors that will appear from the model validation inside the summary will be the validation errors that are associated with the model and not a specific property of that model.
- In other words, if the users do not fill out their username, but the username is required, and there will be a validation error for that specific property.
- But you can also generate model errors that are not associated with a specific property and they will appear in this ValidationSummary.
- Inside a <form> tag we have labels and inputs for all of the different fields that we have in our ViewModel.
- We need a label for the Username, an input for the Username and also validation messages for the Username.
- The other two properties that we need the user to enter are same and that will have a label and input and a span for the Password and a label and an input and a span for ConfirmPassword.

- We don't need to specify the input types for the Password and ConfirmPassword, because the **asp** for tag helper will make sure to set that input type as a password for us.

- At the end, we need to have the button that says **Register.** When the user clicks on this button, we will submit the form back to the controller.

In the AccountController, we also need to implement an HttpPost Register action method. Let us go back to the AccountController and add the following Register action as follows.

```
[HttpPost]
public IActionResult Register (RegisterViewModel model)
{


}
```

This action method will return an IActionResult. This will receive a RegisterViewModel. Now, we need to interact with the Identity framework to make sure that the user is valid, to tell the Identity framework to create that user, and then since they have just created the account, go ahead and log them in. We will look at implementing all these steps in the next chapter.

28. ASP.NET Core — Create a User

In this chapter, we will discuss how to create user. To proceed with this, we need to interact with the Identity framework to make sure that the user is valid, then create that user, and then go ahead and log them in.

- There are two core services of the Identity framework, one is the **UserManager**, and the other is the **SignInManager**.

- We need to inject both of these services into our controller. With this, we can call the appropriate APIs when we need to create a user or sign in a user.

- Let us add private variables for SignInManager and UserManager and then add a constructor in your AccountController, which will take two parameters UserManager of type User and a SignInManager of type User.

```
private SignInManager<User> _signManager;
private UserManager<User> _userManager;

public AccountController(UserManager<User> userManager,
                         SignInManager<User> signManager)
{
    _userManager = userManager;
    _signManager = signManager;
}
```

- We will continue with the POST action method of AccountController and one of the first checks that we should always make inside the post action is to check if our ModelState is valid.

- If the ModelState is valid, then we know the user gave us a username and a password and confirmed the password; if not, we need to ask them to provide the correct information.

- Here is the implementation of the Register action.

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new User { UserName = model.Username };
        var result = await _userManager.CreateAsync(user,
model.Password);
```

```
        if (result.Succeeded)
        {
            await _signManager.SignInAsync(user, false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
        }

    }
    return View();
}
```

- If our **ModelState** is valid, we need to talk to the Identity framework. We also need to create a new instance of our User entity and copy our input **model.Username** into the **UserName** property of the User entity.

- But, we are not going to copy the password because there is no place to store the plain text password in the User entity. Instead, we will pass the password directly to the Identity framework, which will hash the password.

- So we have a userManager. Create an **Async method** where we have to pass the Username, so that we can save the password for that user.

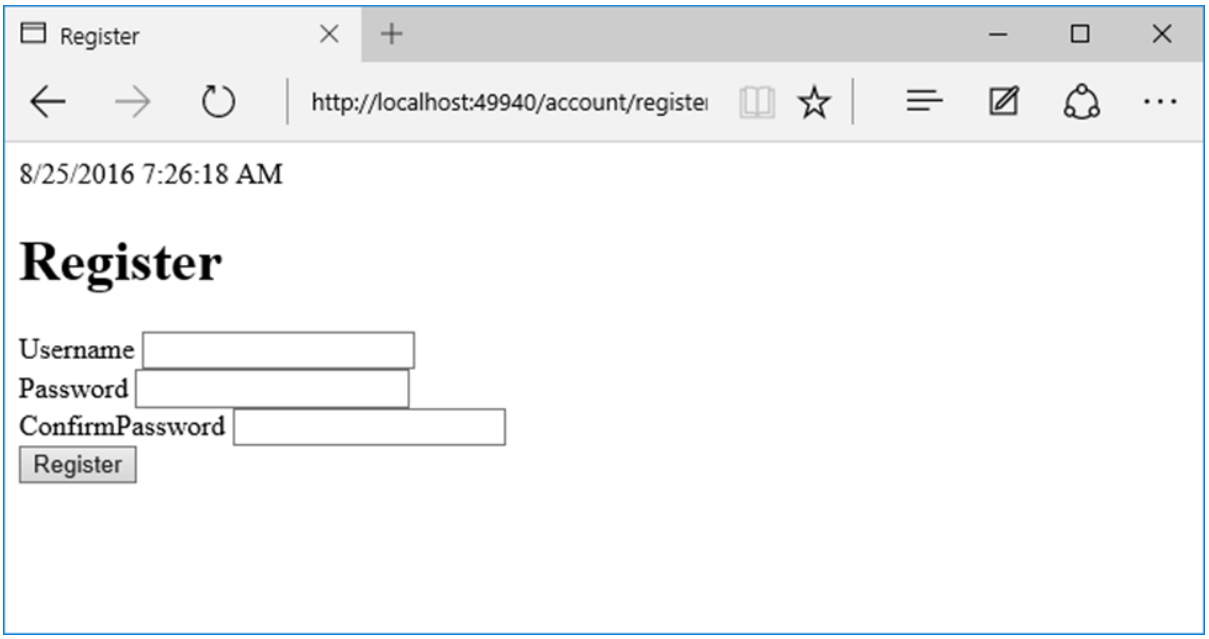
- This Async method returns a result that tells us if the instance was a success or a failure and if it failed, it will give us some of the possible reasons why it failed.

- If the result is successful, we can sign in the user that just created an account and then ask the SignInManager to sign this user. Now, redirect the user back to the home page and you will now be authenticated.

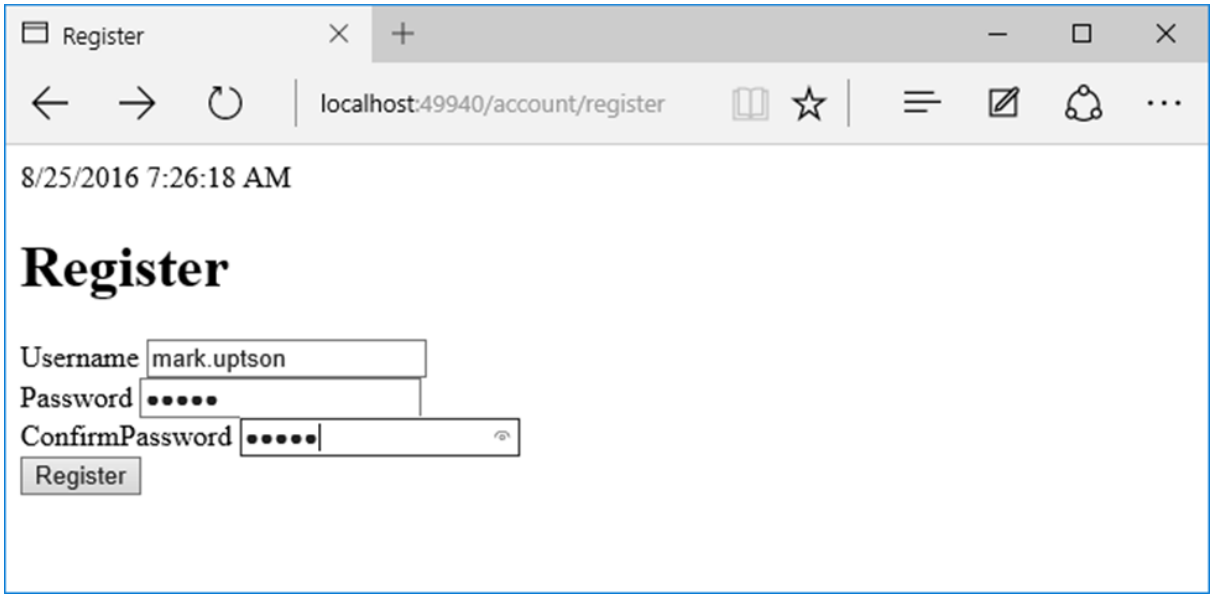
- If the result was not successful, then we should try to tell the user why, and the result that comes back from the UserManager has a collection of errors that we can iterate and add those errors into ModelState. These errors will be available in the view for the tag helpers like the validation tag helpers, to display information on the page.

- In the ModelState.AddModelError, we can provide a key to associate an error with a specific field. We will also use a blank string and add the description of the error that was provided.

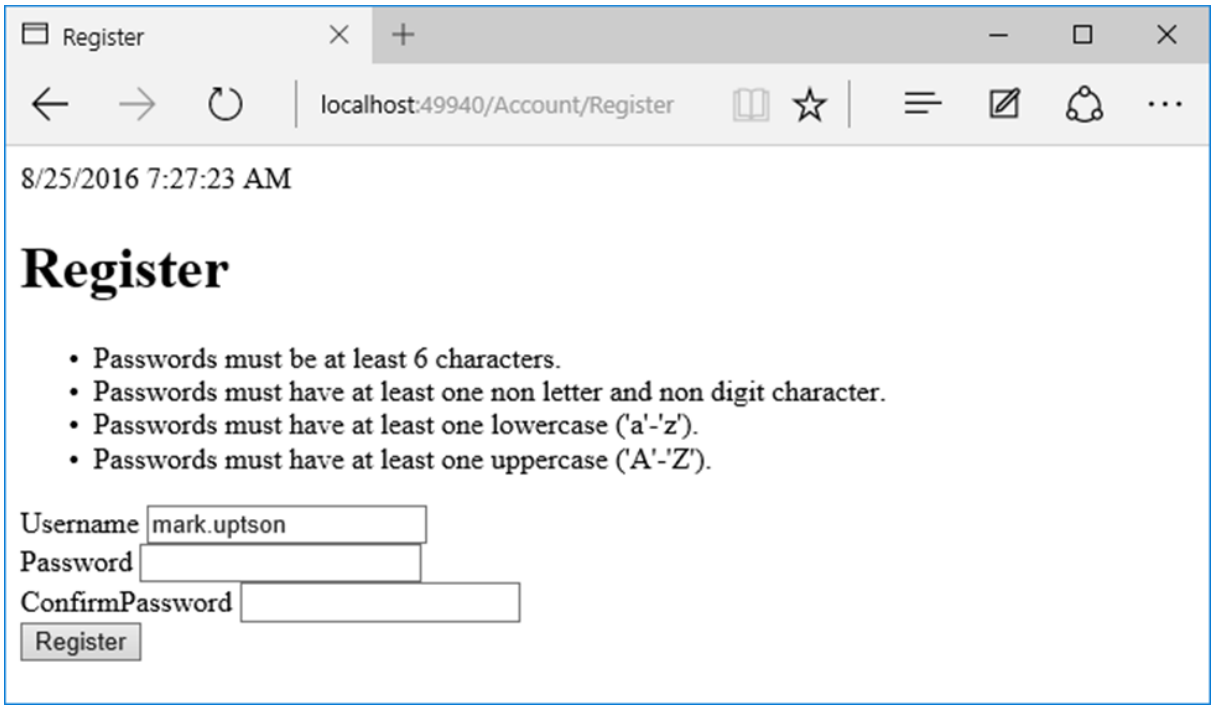
Let us save all the files and run the application and go to **/account/register**.



Let us enter a username and a very simple 5-character password.



Now, click the Register button.

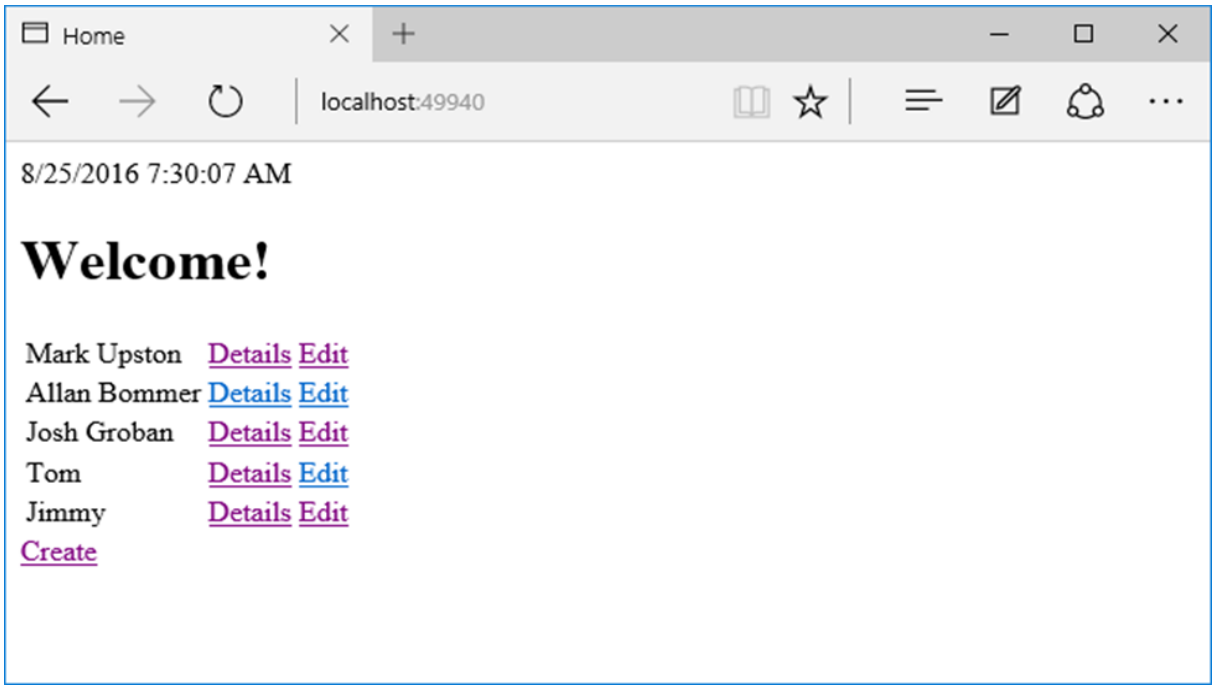


By default, the Identity framework tries to enforce some rules around passwords.

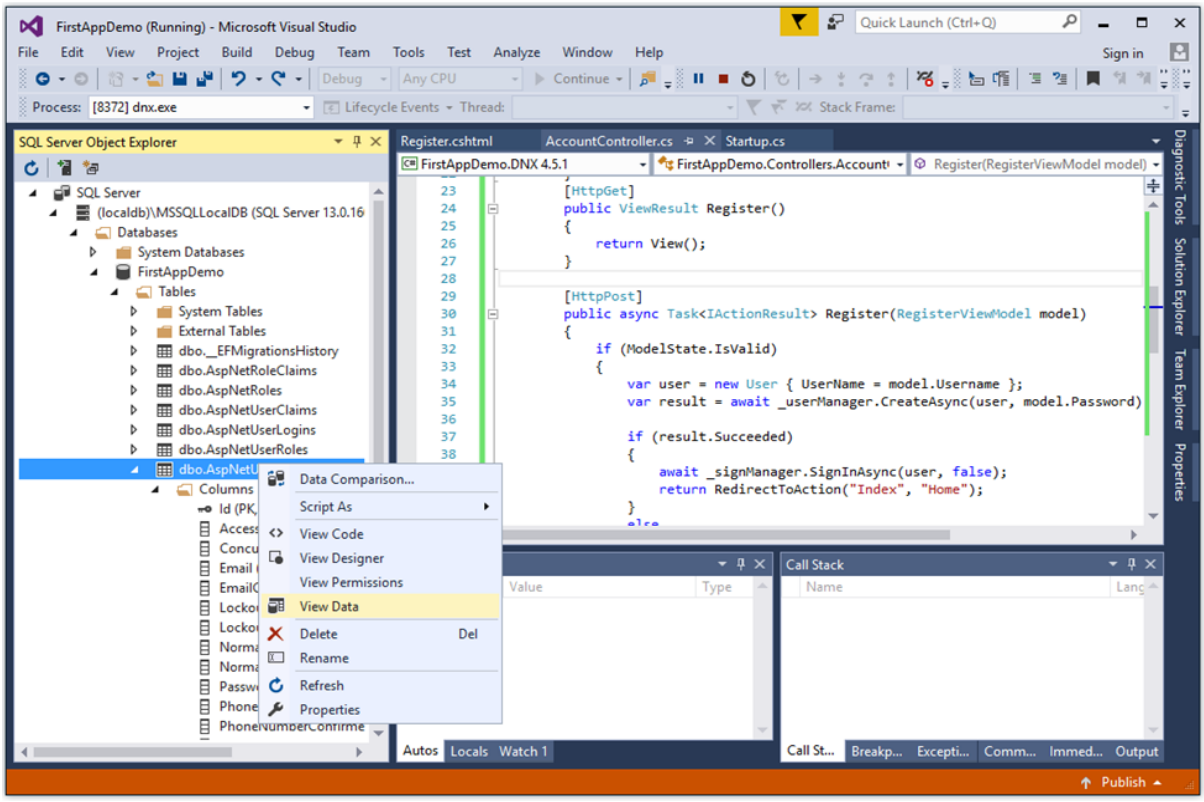
The passwords have to have at least 6 characters, one character has to be in lowercase, one has to be in uppercase, and there has to be one non-digit character.

The reason these errors appear here is because we have a validation summary on the page that is picking up the errors that come back from the **userManager.CreateAsync** result.

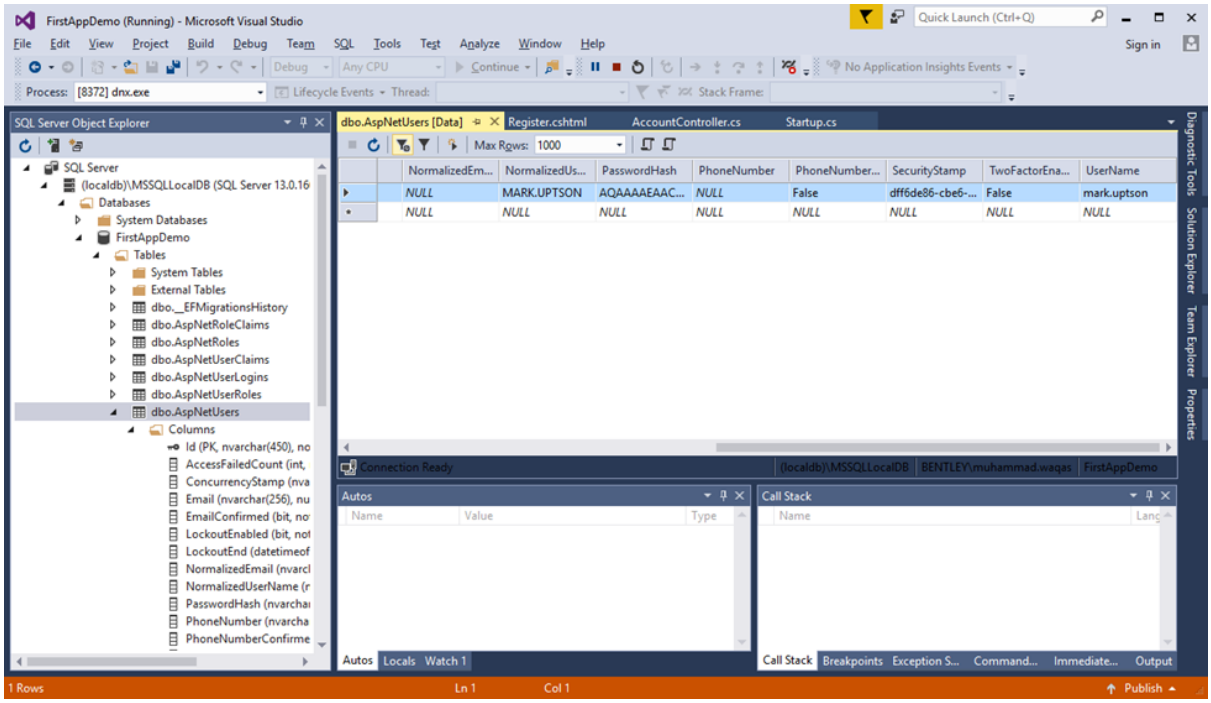
Now that we know a little more about what the password rules are, let us try and create a sufficiently complex password and click Register.



You will now see the home page. This means that the operation worked. Let us now go to the SQL Server Object Explorer.



Right-click on the **dbo.AspNetUsers** table and select the **View Data**.



You can now see that the user was created successfully and you can also see a new record in the Users table. You can also see a hashed password value as well as a username and that is the username that we registered with **mark.upston**.

29. ASP.NET Core — Log in and Log Out

In this chapter, we will discuss the login and logout feature. Logout is rather simple to implement as compared to login. Let us proceed with the Layout view because we want to build a UI that has some links. This will allow a signed in user to log out and also display the username.

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @DateTime.Now
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

- For an anonymous user, we will show a login link.
- All the information you need to build this UI is available from the Razor view context.
- First, let us add the namespace **System.Security.Claims** in your layout view.

```
@using System.Security.Claims
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
```

```
    <div>
        @if (User.IsSignedIn())
        {
            <div>@User.GetUserName()</div>
            <form method="post" asp-controller="Account" asp-
controller="Logout">
                <input type="submit" value="Logout"/>
            </form>
        }
        else
        {
            <a asp-controller="Account" asp-action="Login">Login</a>
            <a asp-controller="Account" asp-action="Register">Register</a>
        }
    </div>
    <div>
        @DateTime.Now
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

- There is a User property that is available inside every Razor view and we want to build a UI that will display the logged in user's name. An extension method **IsSignedIn** is also available here.

- We can invoke this method and if it returns true, this is where we can place some markup to display the username, display a logout button.

- Now if the user is signed in, we can display the user's username using the helper method **GetUserName**.

- We will have to build a logout button inside a form, which will be posted to the web server. This has to be done as it will create certain unsavory conditions, if you allow a simple GET REQUEST to allow a user out.

- We will force this to be a post, and when the user submits this form, all we need to do is hit on the Logout action, which we will implement through the AccountController, and logout the user.

- If the user is not signed in and we have an anonymous user, then we need to show a link that will go to the AccountController, specifically to the Login action, and it can display the text Login.

- We also need to add a link for new users to register and go directly to the Register page.

Let us now go to the AccountController and implement the logout action first as in the following program.

```
[HttpPost]
public async Task<IActionResult> Logout()
{
    await _signManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
```

- This action responds only to the HttpPost. This is an async action. We will have to call another asynchronous method on the Identity framework.
- We can return a task of IActionResult and the action is named Logout.
- All we need to do to logout is to await the **SignInManager's SignOutAsync** method.
- The user context has changed now; we now have an anonymous user. The view will be redirected to the home page and we will go back to the list of employees.

Let us now proceed and build our Login feature. Here, we will need a pair of actions, one that responds to an HttpGet request and displays the form that we can use to log in, and one that responds to an HttpPost request.

To begin with, we will need a new ViewModel to pull the login data because logging in is very different from registering. So, let us add a new class and call it **LoginViewModel**.

```
public class LoginViewModel
    {
        public string Username { get; set; }

        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Display(Name ="Remember Me")]
        public bool RememberMe { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

- When the users log in, they have to provide some information like the username, password.

- The third piece of information must be login UIs. These come with a little checkbox that says — "Do you want to remember me". This is the choice between do we want a session cookie, or do we want a more permanent cookie.

- To allow this feature we have added a Boolean property **RememberMe** and we have used a Display annotation. Now when we build a label, the text **Remember Me** gets displayed with a space.

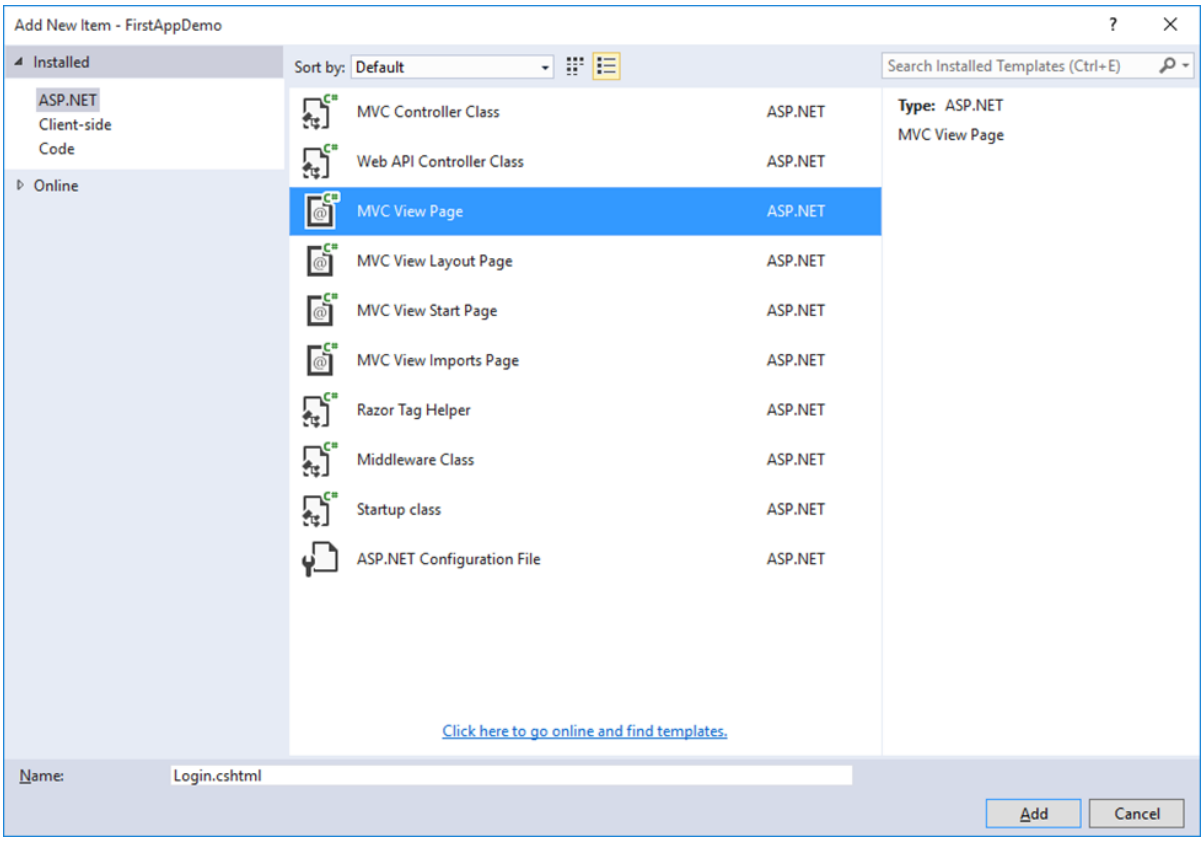
- The last information that we actually want as part of this ViewModel is to have a property that will store the ReturnUrl.

Let us now add the Login action that will respond to the Get request as shown in the following program.

```
[HttpGet]
public IActionResult Login(string returnUrl = "")
{
    var model = new LoginViewModel { ReturnUrl = returnUrl };
    return View(model);
}
```

- We will take the **returnUrl** as a parameter that is in the query string.
- The **returnUrl** might not always be there. Let us have an empty string as the default.

We will now have a new view by adding a new MVC View Page in the **Views > Account** folder.



In the middle pane, select the MVC View Page and call it Login.cshtml and then Click on the Add button. Let us add the following code in the Login.cshtml file

```
@model LoginViewModel
@{
    ViewBag.Title = "Login";
}

<h2>Login</h2>

<form method="post" asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@Model.ReturnUrl">
    <div asp-validation-summary="ValidationSummary.ModelOnly"></div>
    <div>
        <label asp-for="Username"></label>
        <input asp-for="Username" />
        <span asp-validation-for="Username"></span>
    </div>
    <div>
        <label asp-for="Password"></label>
        <input asp-for="Password" />
```

```
        <span asp-validation-for="Password"></span>
    </div>
    <div>
        <label asp-for="RememberMe"></label>
        <input asp-for="RememberMe" />
        <span asp-validation-for="RememberMe"></span>
    </div>
    <input type="submit" value="Login" />
</form>
```

- In this login view, we have set the title of the page to **Login** and then we have a form that will post to the **AccountLogin** action.
- We need to use a tag helper, **asp-route-returnurl**, to make sure that the ReturnUrl is there in the URL that the form posts back to.
- We need to send that ReturnUrl back to the server so that if the user does successfully log in, we can send it over to the place they were trying to get to.
- Anything that you add after asp-route-, id or returnurl, whatever you have there, that will go into the request somewhere, either into the URL path or as a query string parameter.
- We have our **ValidationSummary** and inputs for Username, Password, and RememberMe, and then we have a Submit button.

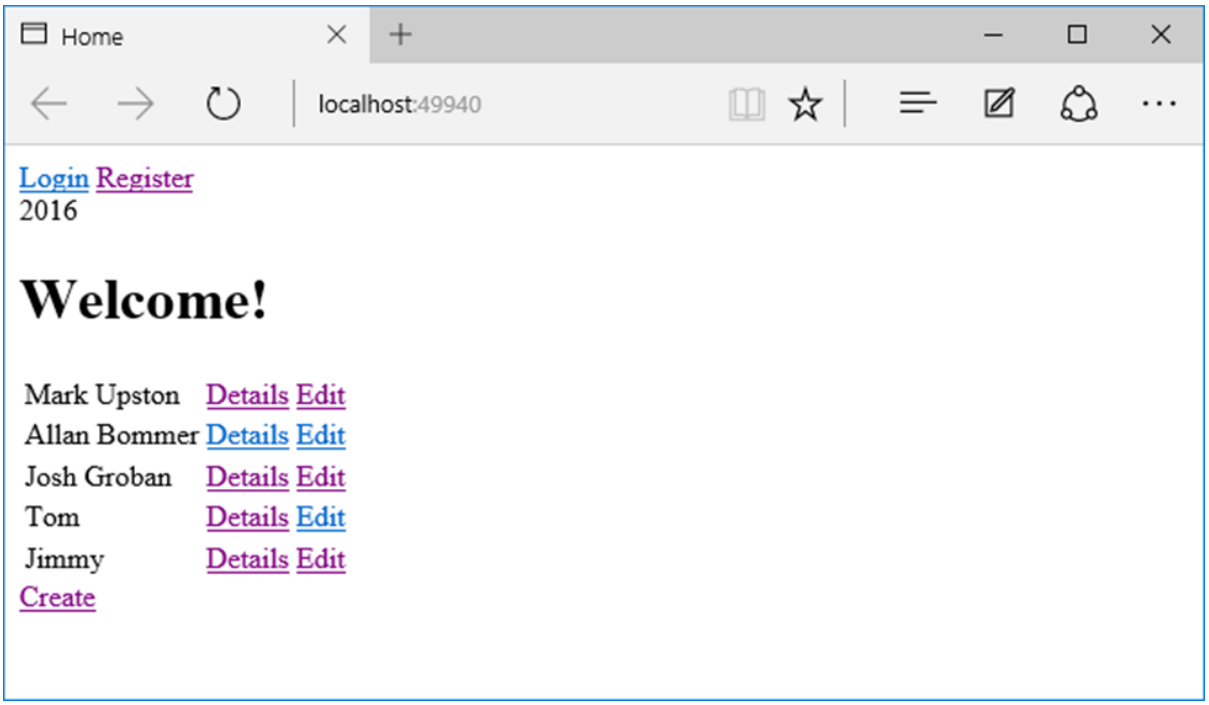
Let us now go back to the **AccountController**, and implement the Post action. This action that responds to HttpPost. This will be an Async method because we will need to call into the Identity framework and return a task or IActionResult.

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signManager.PasswordSignInAsync(model.Username,
                                                            model.Password,
                                                            model.RememberMe,
                                                            false);
        if (result.Succeeded)
        {
            if (!string.IsNullOrEmpty(model.ReturnUrl) &&
Url.IsLocalUrl(model.ReturnUrl))
            {
                return Redirect(model.ReturnUrl);
```

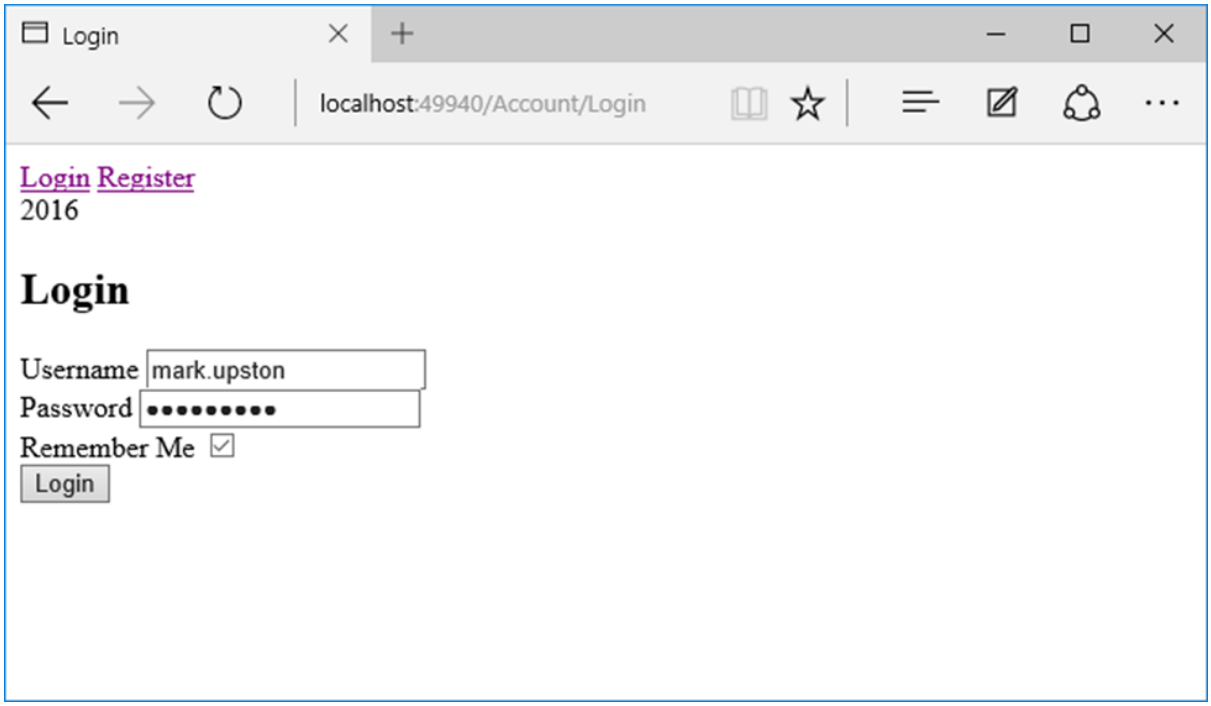
```
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
    }
    ModelState.AddModelError("","Invalid login attempt");
    return View(model);
}
```

- We call this Login, and now we expect to receive a LoginViewModel.
- We need to to check if the ModelState is valid. If it is valid, then sign in the user by calling an API on the SignInManager.
- The **PasswordSignInAsync** method will return a result and if the result succeeded, we know the user has logged in successfully.
- We also have a return URL. If it is a valid local Url, we will be redirected to the return URL.
- If the user has just logged in and does not have any specific place to go, we will redirect the user to the Index action of the HomeController.
- We might be in a situation where the user provides an invalid password or an invalid username. We also need to add a model error that prompts if there is an Invalid login attempt. This helps the user to know if something went wrong.

Let us now save everything and run the application.

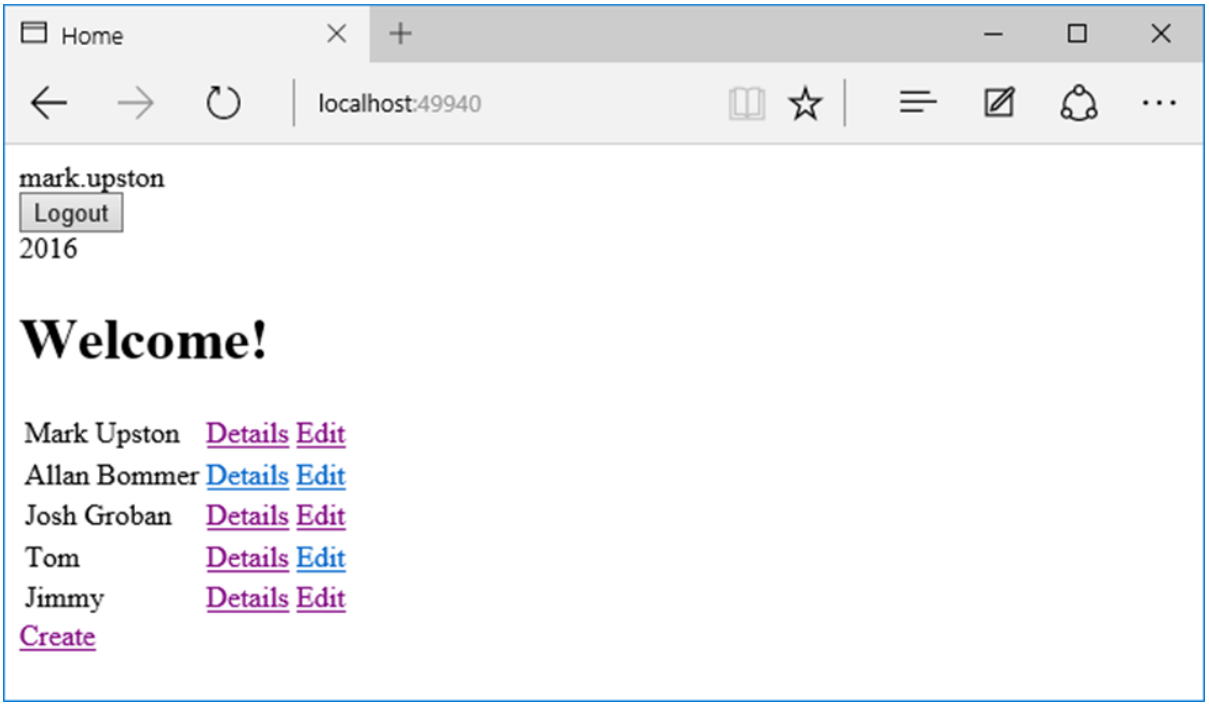


We now have Login and Register links. Let us click on the Login link.

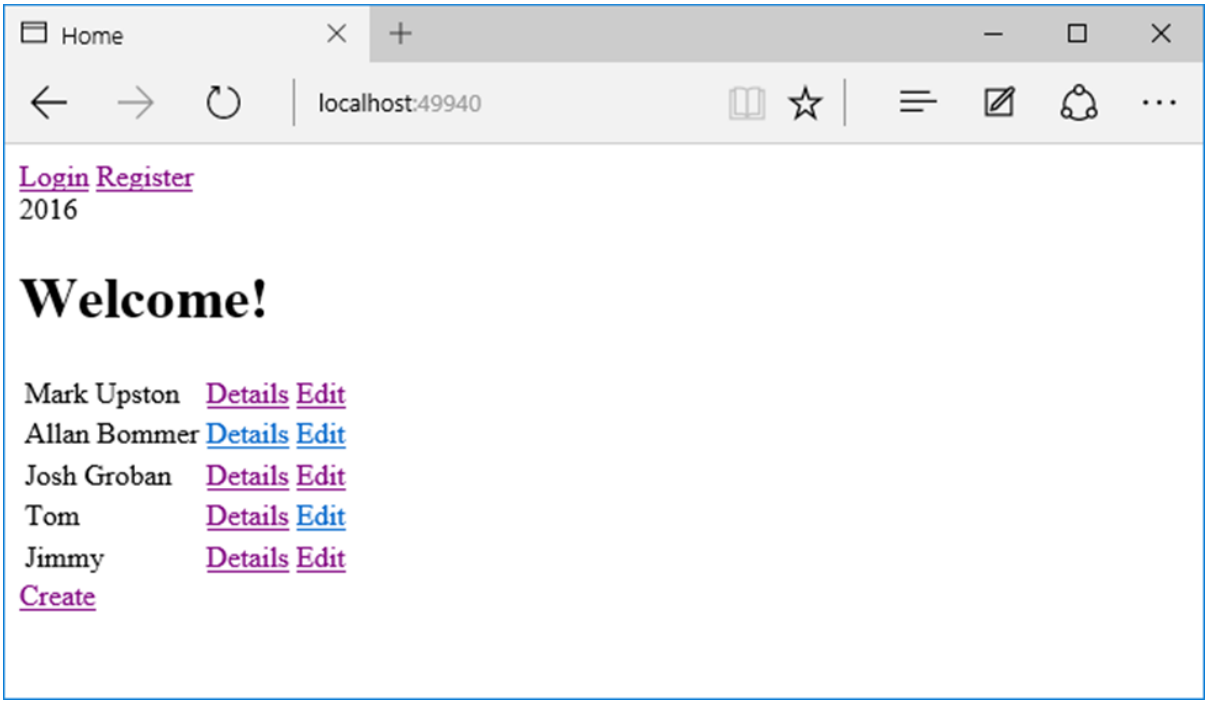


Let us login with the user that we created in the previous chapter by specifying the **Username** and **Password** and check the **Remember Me** check box.

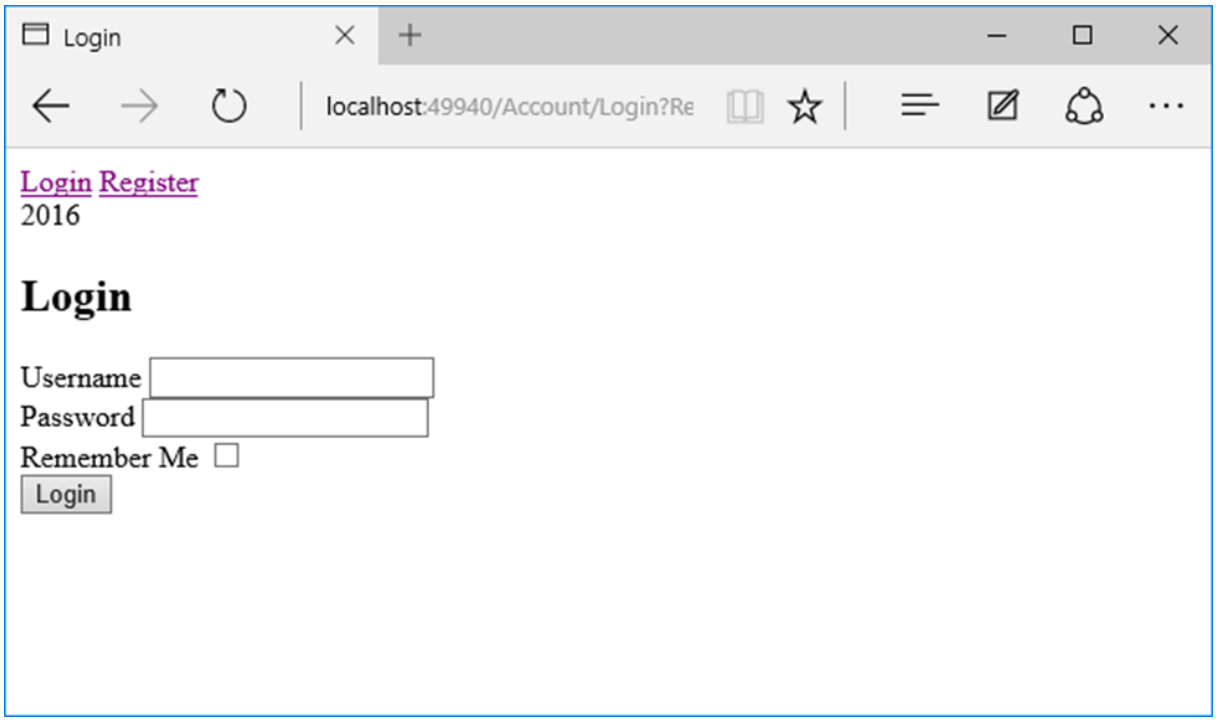
When you click on the Login button, the browser will ask you if you would you like to save your password for the localhost. Let us click on the Yes button.



Now, let us logout by clicking on the Logout button.

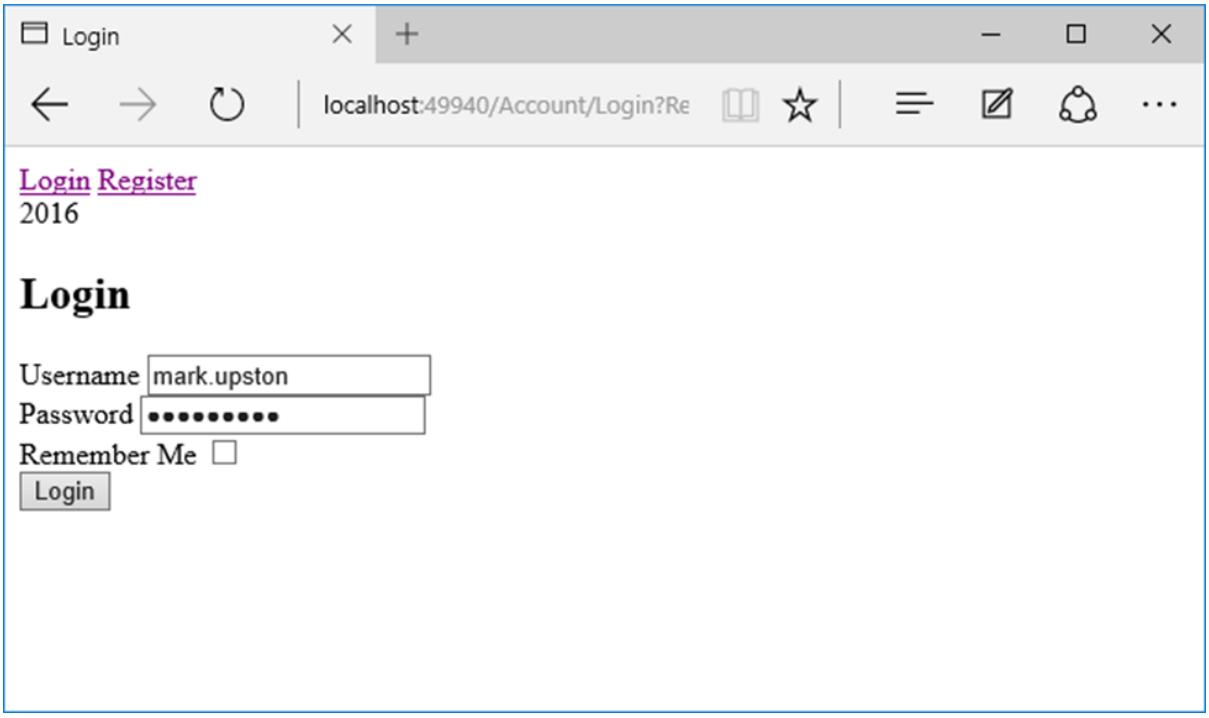


As an anonymous user, let us go and try to edit employee details.

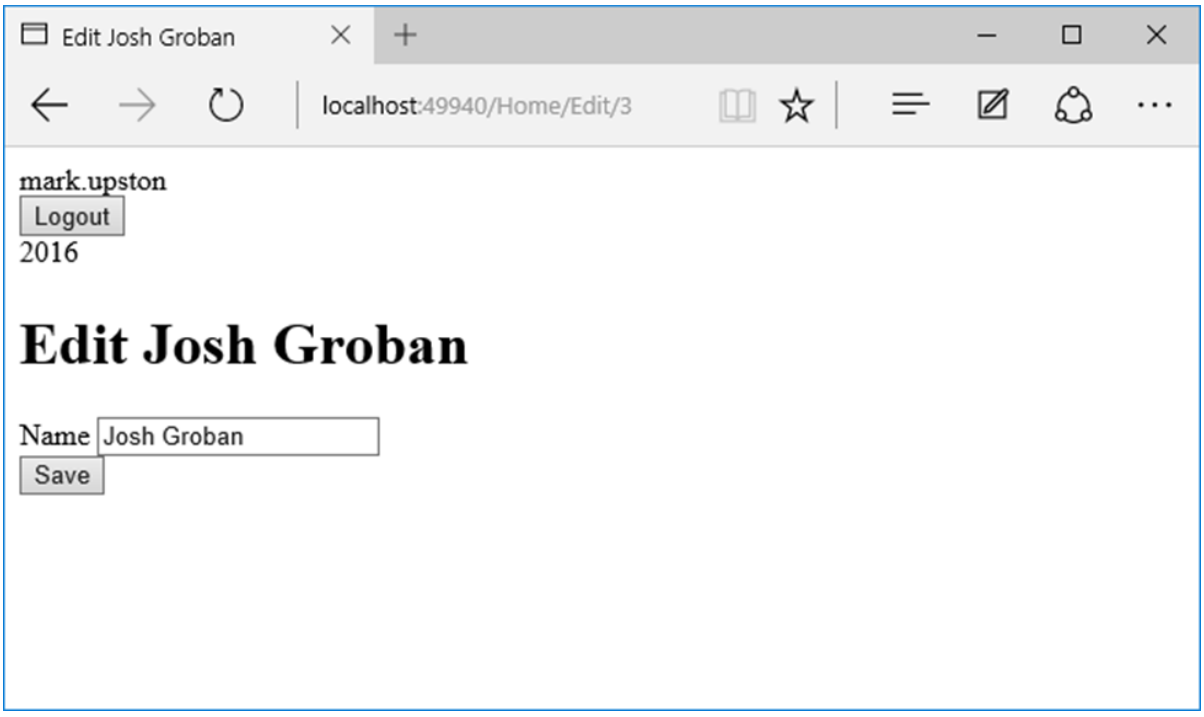


You can see now that we have been redirected to the **Login view**.

Let us log in with your username and password and check the Remember Me check box.



Now, click the Login button.



You can now see that we are directed to the URL that we want to edit. This is because we processed the return URL appropriately.