**ASP.NET Core 1.1**

ASP.NET Core is a new open-source and cross-platform .NET framework for building modern cloud-based web applications on Windows, Mac, or Linux. ASP.NET Core can be used for building modern cloud-based Internet-connected applications, such as web apps, IoT apps and mobile backends. It was architected to provide an optimized development framework for apps that are deployed to the cloud or run on-premises.

**Why use ASP.NET Core?**

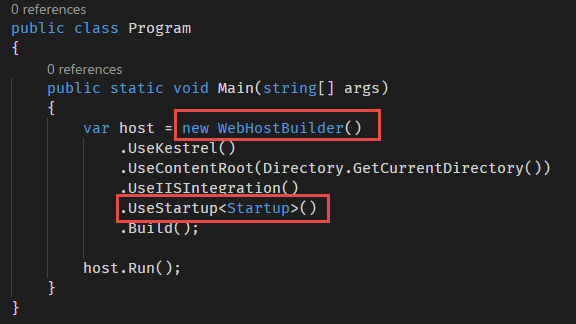
ASP.NET Core is a significant redesign of ASP.NET, with architectural changes that result in a leaner and modular framework. ASP.NET Core is not based on ***System.Web.dll***. It is based on a set of granular and well-factored ***NuGet packages***. This allows you to optimize your app to include just the NuGet packages you need. The benefits of a smaller app surface area include ***tighter security, reduced servicing, improved performance, and decreased costs in a pay for what you use model***.

**ASP.NET Core provides the following improvements compared to ASP.NET:**

* A unified story for building web UI and web APIs.
* Integration of modern client-side frameworks and development workflows.
* A cloud-ready environment-based configuration system.
* Built-in dependency injection.
* A light-weight and modular HTTP request pipeline.
* Ability to host on IIS or self-host in your own process.
* Built on .NET Core, which supports true side-by-side app versioning.
* Ships entirely as NuGet packages.
* New tooling that simplifies modern web development.
* Build and run cross-platform ASP.NET Core apps on Windows, Mac, and Linux.
* Open-source and community-focused.

***Startup Class and Request pipeline***

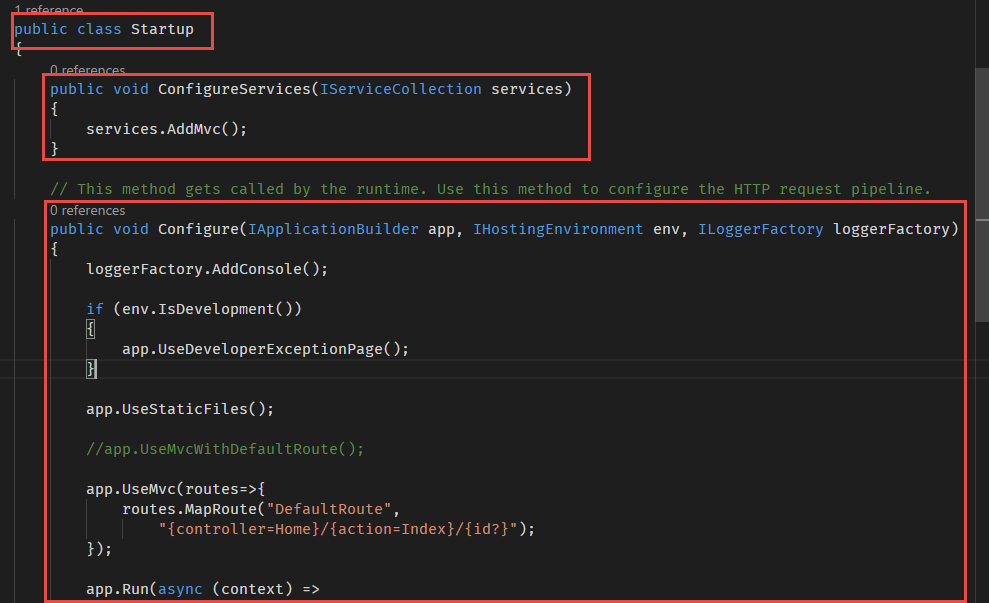
The Startup class configures the request pipeline that handles all requests made to the application. ASP.NET Core apps require a Startup class. By convention, the Startup class is named "Startup". You specify the startup class name in the Main programs ***WebHostBuilderExtensions UseStartup<TStartup>*** method



You can define separate Startup classes for different environments, and the appropriate one will be selected at runtime. If you specify ***startupAssembly*** in the ***WebHost configuration*** or ***options***, hosting will load that ***startup assembly*** and search for a ***Startup*** or ***Startup[Environment]*** type.

**Startup class constructor and DI**

The Startup class constructor can accept dependencies that are provided through dependency injection. You can use ***IHostingEnvironment*** to set up configuration sources and ***ILoggerFactory*** to set up logging providers. The Startup class must include a ***Configure*** method and can optionally include a ***ConfigureServices*** method, both of which are called when the application starts.



**Need for Configure method in Startup class**

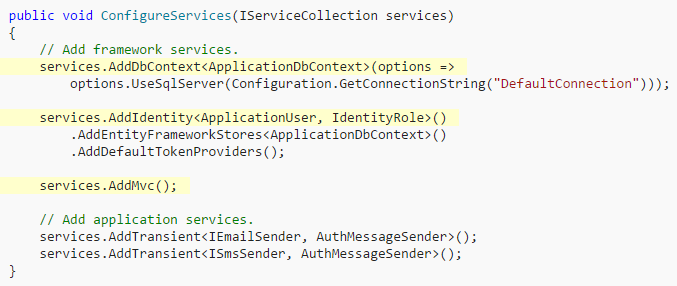
The Configure method is used to specify how the ASP.NET application will respond to HTTP requests. The request pipeline is configured by adding middleware components to an IApplicationBuilder instance that is provided by dependency injection. Each ***Use extension method*** adds a ***middleware component*** to the request pipeline.



**ConfigureServices method in Startup class**

The ***ConfigureServices*** method is optional; but if used, it's called before the Configure method by the runtime. Some services need to be added to the Services collection before they are configured in the request pipeline. Eg: MVC middleware.

You can use the ***Add[Service]*** extension method to add the services to the ***IServiceCollection.***



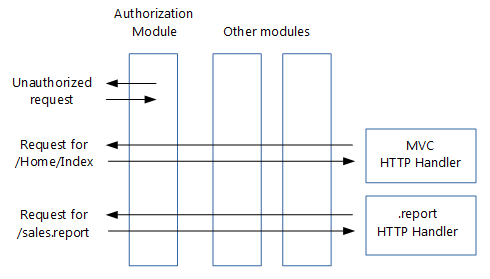
**Middleware in ASP.NET Core**

Middleware is software that is 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.

Request delegates are configured using ***Run***, ***Map***, and ***Use*** extension methods on the ***IApplicationBuilder*** instance that is passed into the ***Configure*** method. An individual request delegate can be specified in-line as an anonymous method (called ***in-line middleware***), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are ***middleware***, or ***middleware components***. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline, or short-circuiting the chain if appropriate.

**ASP.NET Core Middleware Vs Modules and Handlers**

ASP.NET Modules and Handlers



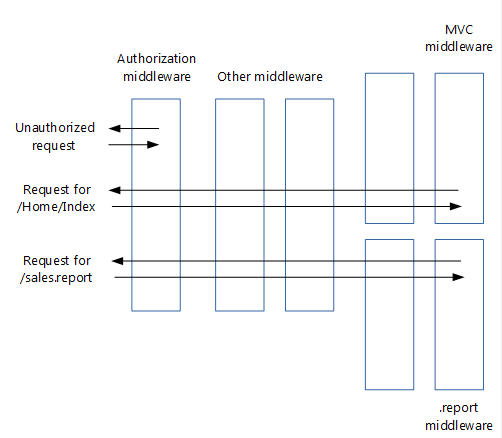
**Handlers are:**

1. Classes that implement **IHttpHandler**
2. Used to handle requests with a given file name or extension, such as .report
3. Configured in **Web.config**

**Modules are:**

1. Classes that implement **IHttpModule**
2. Invoked for every request
3. Able to short-circuit (stop further processing of a request)
4. Able to add to the HTTP response, or create their own
5. Configured in **Web.config**

**Middleware in Action**



1. Modules, handlers, Global.asax.cs, Web.config (except for IIS configuration) and the application life cycle are gone
2. The roles of both modules and handlers have been taken over by middleware
3. Middleware are configured using code rather than in Web.config
4. Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.
5. Invoked in principle for every request
6. Able to short-circuit a request, by not passing the request to the next middleware
7. Able to create their own HTTP response
8. Order of middleware is based on the order in which they are inserted into the request pipeline, while order of modules is mainly based on application life cycle events
9. Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses

**Create ASP.NET Core 1.1 MVC Project from Scratch**

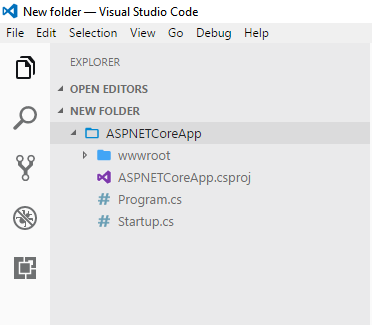
**Steps**

1. Create a new folder in your system where you want to create the project and Open VS Code and set the new folder as your working directory.
2. Open the VS Code integrated Terminal, by pressing Ctrl+` or View > Integrated Terminal.
3. Run the following command to create a new “Empty web application”.

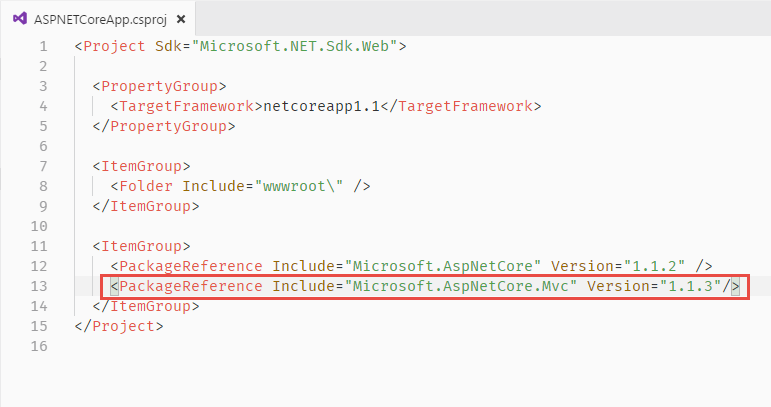
**dotnet new web <ProjectName>**

**eg: dotnet new web ASPNETCoreApp**

1. It creates a set of files in the directory, you can see all files and folders in the VS Code.



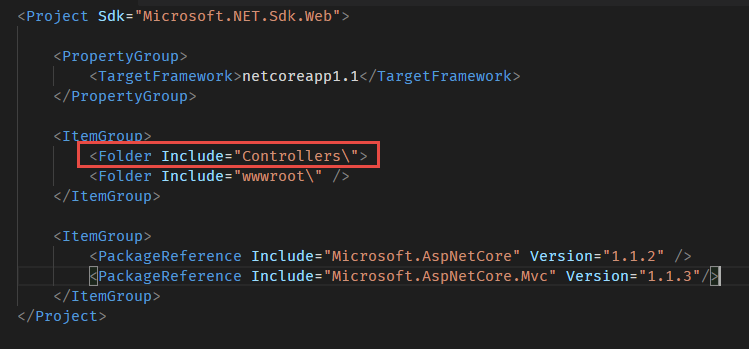
1. Now you can start adding dependencies to the project. To Create an MVC application you need to add MVC library to the project.
2. To add a new library, open the .**csproj** file and add a new **PackageReference** element under **ItemGroup**. Type the Name of the package/library as the value of the **Include** attribute and version number as the **version** attribute value.



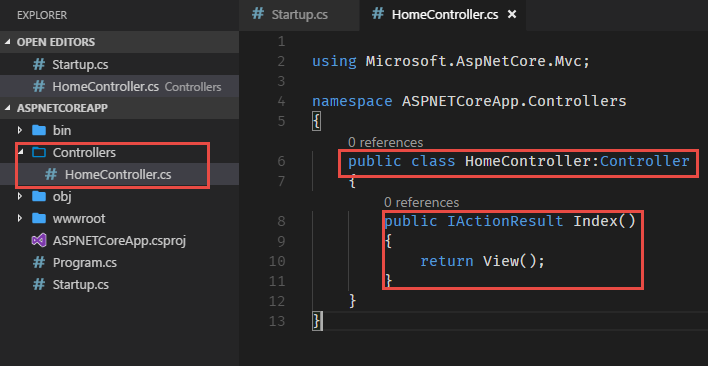
1. Run the ***dotnet restore*** command to restore the libraries.
2. Now you can add the MVC middleware to the **startup.cs** file. Open the **startup.cs** file and add the MVC middleware line to **ConfigureServices()** method.
3. You also need to configure the MVC middleware in the **Configure**() method.



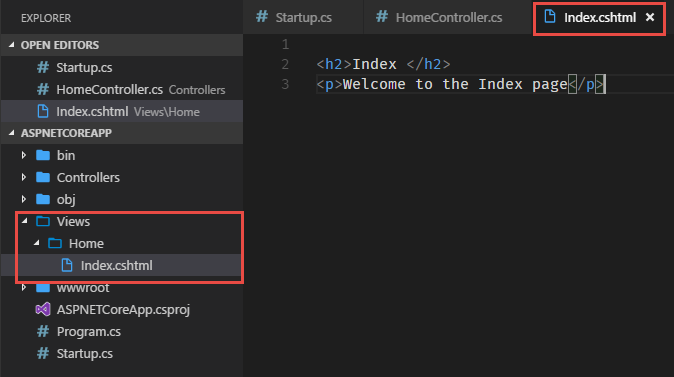
1. You have configured MVC with default route.
2. Now you can add a new MVC controller to the project. Create a new folder called “***Controllers***” in the root folder.
3. Add a new file called “***HomeController.cs***” to the ***Controllers*** folder.
4. You also need to add the folder name in the Serving folders list in the .csproj file.



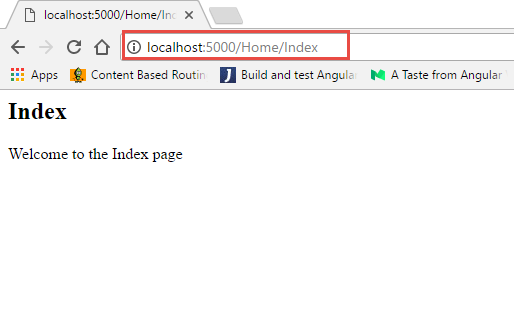
1. Add an Index() method to the HomeController class.



1. Now you need to create the views for the ***Index action***. To do so, Create a new folder called “***Views***” inside the root folder. Add a subfolder with the same name as controller, ie. ***Home*** and create ***Index.cshtml*** file inside the ***Home*** folder.



1. Run the project by running the command “***dotnet run***”. It runs the project in port number 5000, ie. <http://localhost:5000>. Open the browser and navigate to the address <http://localhost:5000>.
2. You will get the default message, configured in the ***Startup.cs*** file. To Run the ***HomeController > Index*** action type <http://localhost:5000/Home/Index> and press enter. You will get the ***index view*** in the browser.



1. Well, you have successfully completed your first ASP.NET Core application.

**Configuring Default Controller and Action for MVC**

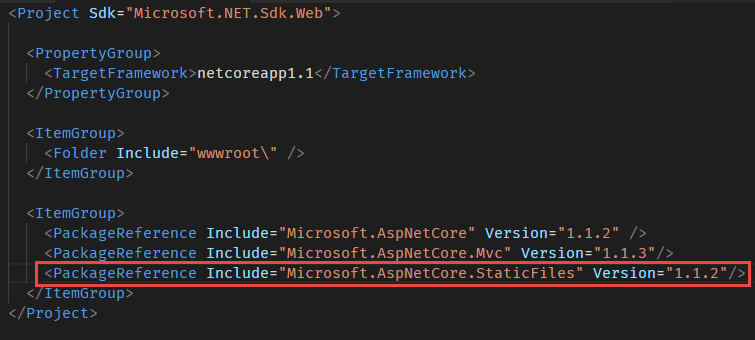
1. Now we need to configure the Default controller and Action for your MVC application.
2. To add default route, Open the ***Startup.cs*** file and comment the ***app.UseMvcDefaultRoute()*** and add the following line of code with default controller and action names.



1. Now you have configured the default route for MVC. To test this open the browser and type <http://localhost:5000> in the browser. It loads the ***Index*** view in the browser.

**Serving Static files in MVC application**

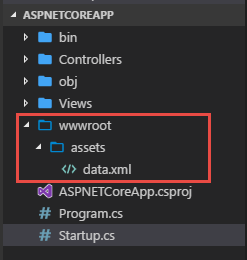
1. You can also configure your application to serve static files, such as images, audio and video files, xml and json files etc.
2. To enable static file support on your application, you need to add a middleware to your application. For that open the .**csproj** file and add the “***Microsoft.ASpNetCore.StaticFiles***” package to the project.



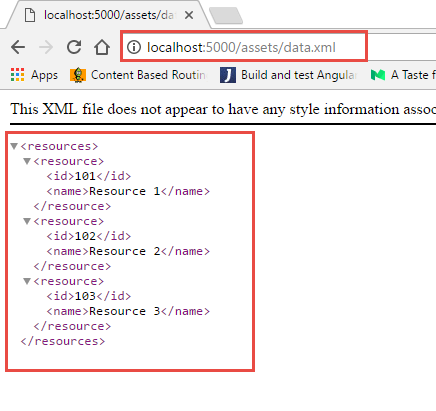
1. Now you need to configure the StaticFiles middleware in the **Configure()** method of **Startup.cs** file.



1. To test the service, create a new folder called “assets” in the wwwroot folder. Copy some images and text files in the assets folder.



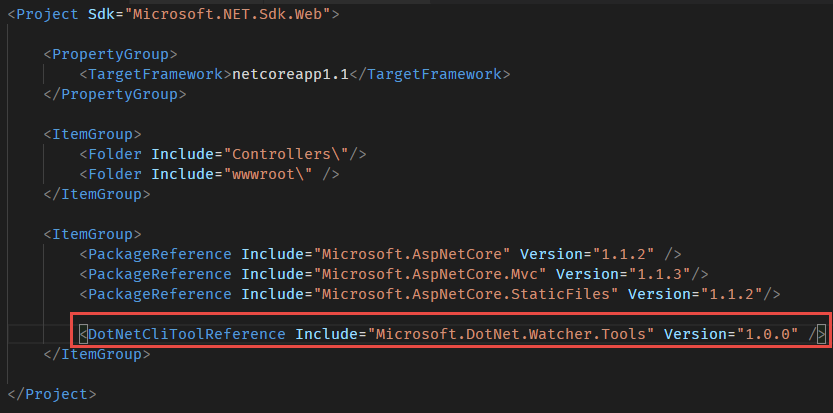
1. Open browser and navigate to the URL, <http://localhost:5000/assets/data.xml> . The XML file will be opened in your browser.



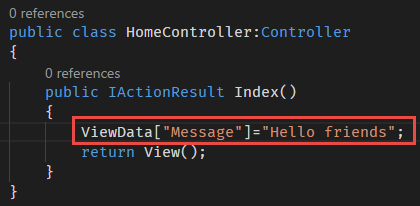
**Adding dotnet watch tool in projects**

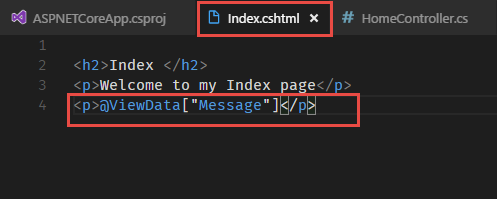
***dotnet watch*** is a tool that runs a dotnet command when source files change. For example, a file change can trigger compilation, tests, or deployment. To enable **dotnet watch** tool in your projects follow the steps.

1. Add “**Microsoft.DotNet.Watcher.Tools**” package to the .**csproj** file as **DotNetCliToolReference** not as **PackageReference**.

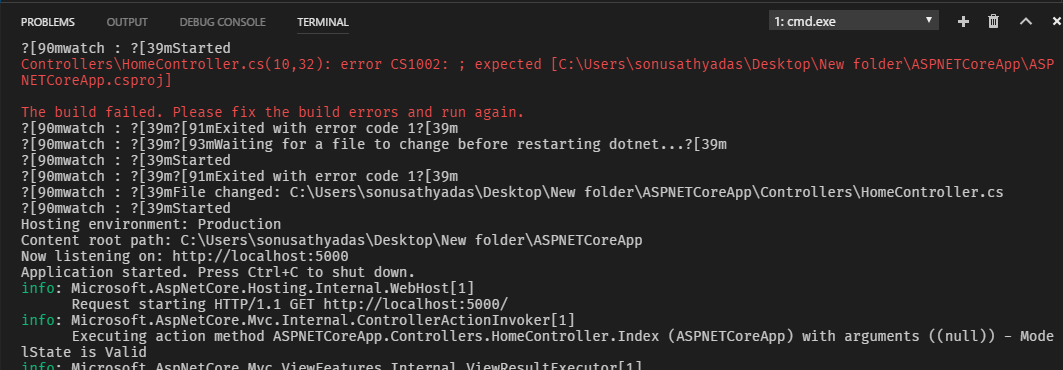


1. Restore the nuget packages using the command “dotnet restore”.
2. Run the project in watch mode by using the command “dotnet watch run”.
3. Open browser and navigate to the project URL. You will see the index page in the browser.
4. Make some changes in the Controller file or/and View files. It will be automatically compiled. You don’t need to stop and re-run the project each time.





1. You can see the compilation results in the console.



**Adding EntityFramework Core in ASP.NET Core**

EF Core 1.1 is the latest version of EF but does not yet have all the features of EF 6.x. Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology. EF Core is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write. EF Core supports many database engines.

**List of database engines supported by EF Core.**

|  |  |
| --- | --- |
| **Database Provider** | **Package** |
| SQLServer | Microsoft.EntityFrameworkCore.SqlServer |
| SQLite | Microsoft.EntityFrameworkCore.SQLite |
| PostgreSql(NpgSql) | Npgsql.EntityFrameworkCore.PostgreSQL |
| IBM Data Server (DB2) | IBM.EntityFrameworkCore (Windows)  IBM.EntityFrameworkCore-lnx(Linux) |
| MySql(Official) | MySql.Data.EntityFrameworkCore -Pre |
| Microsoft SQL Server Compact Edition | EntityFrameworkCore.SqlServerCompact40 |
| InMemory(Testing) | Microsoft.EntityFrameworkCore.InMemory |

1. To work with ORM tools you need to install EntityFrameworkCore in your project. To enable EntityFrameworkCore feature in your project, your need to add the following packages to the .csproj file.

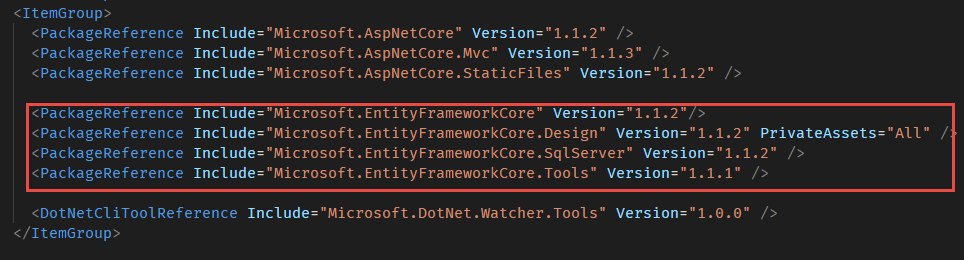
***Microsoft.EntityFrameworkCore***

***Microsoft.EntityFrameworkCore.Design***

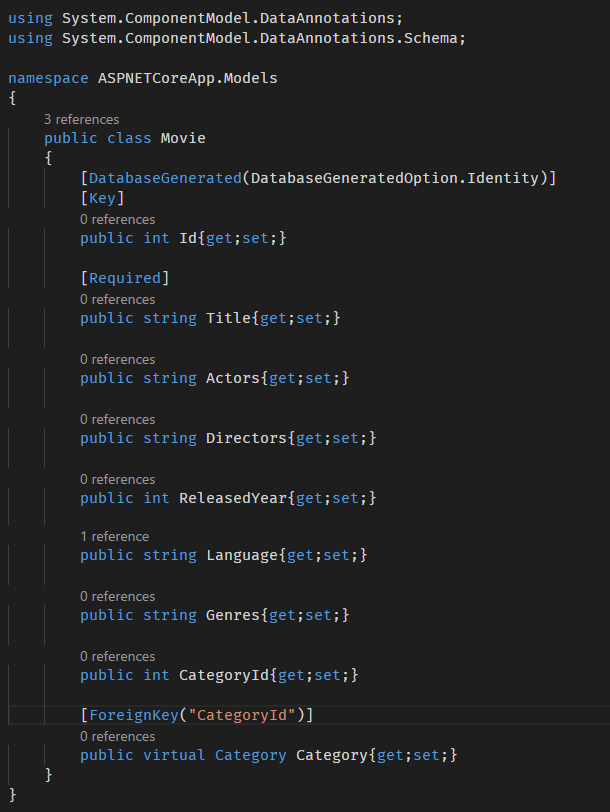
***Microsoft.EntityFrameworkCore.SqlServe***

***Microsoft.EntityFrameworkCore.Tools***

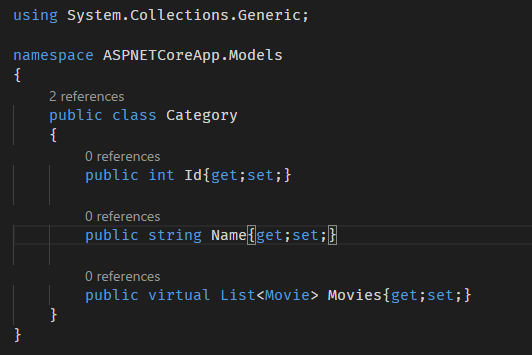
1. Open the .**csproj** file and add the two libraries in the **PackageReference** section.



1. Run the “**dotnet restore**” command to restore packages.
2. Now you need to create necessary **model classes** and **DbContext** classes. To do that, you need to create “Models” folder in the project root.
3. Create a model class named “***Movie.cs***” inside the Models folder.



1. Create another model class “***Category.cs***” in the “**Models**” folder.



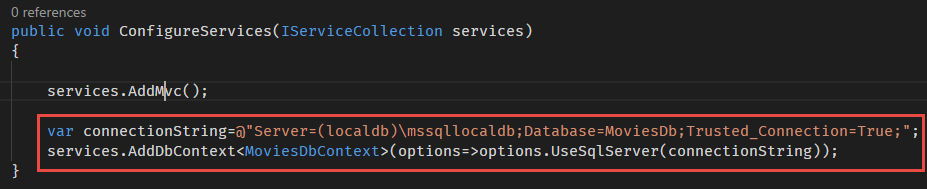
1. Finally, create a DbContext class named “MoviesDbContext.cs”.



1. Now we need to register our context class with Dependency Injection. Add the **AddDbContext<T>()** method to register it as a service in the **ConfigureServices** method of the **Startup.cs** class. Import the namespaces for the classes.

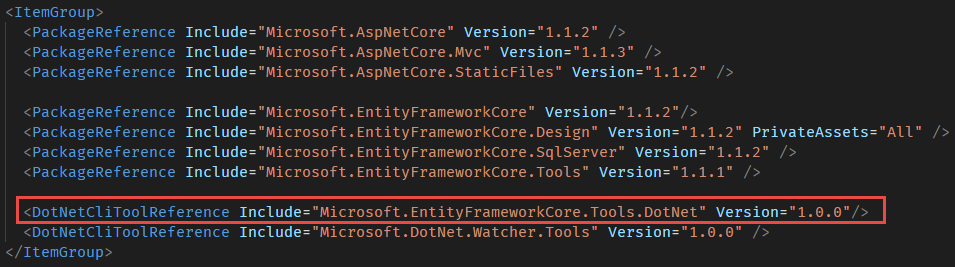


Add the following code in the ConfigureServices() method.

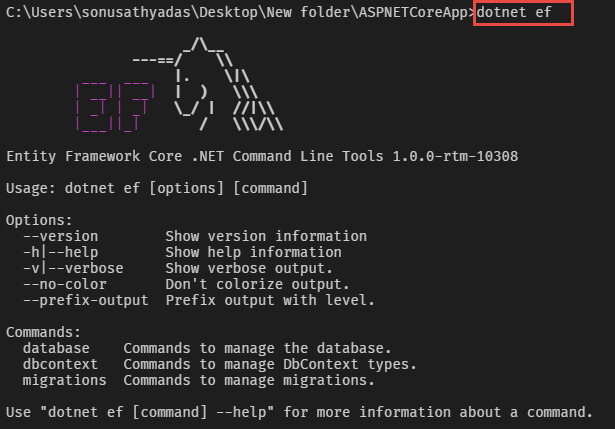


1. Now we need to do database migrations. To enable database migrations on EntityFramework Core, you need to install the following packages as **DotNetCliToolReference**.

**Microsoft.EntityFrameworkCore.Tools.DotNet**

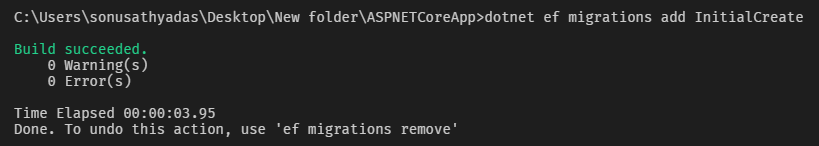


1. Run the “dotnet restore” command to restore packages.
2. Open the terminal and run the “dotnet ef” command to check whether migration tools are installed properly or not.

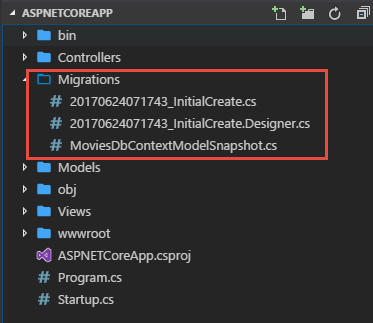


1. Run the following command to enable the initial migration.

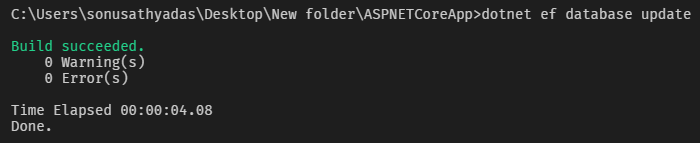
**dotnet ef migrations add InitialCreate**



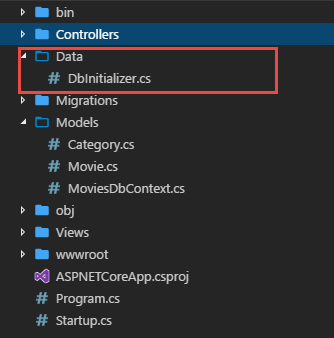
1. It generates a “Migrations” folder in the project.



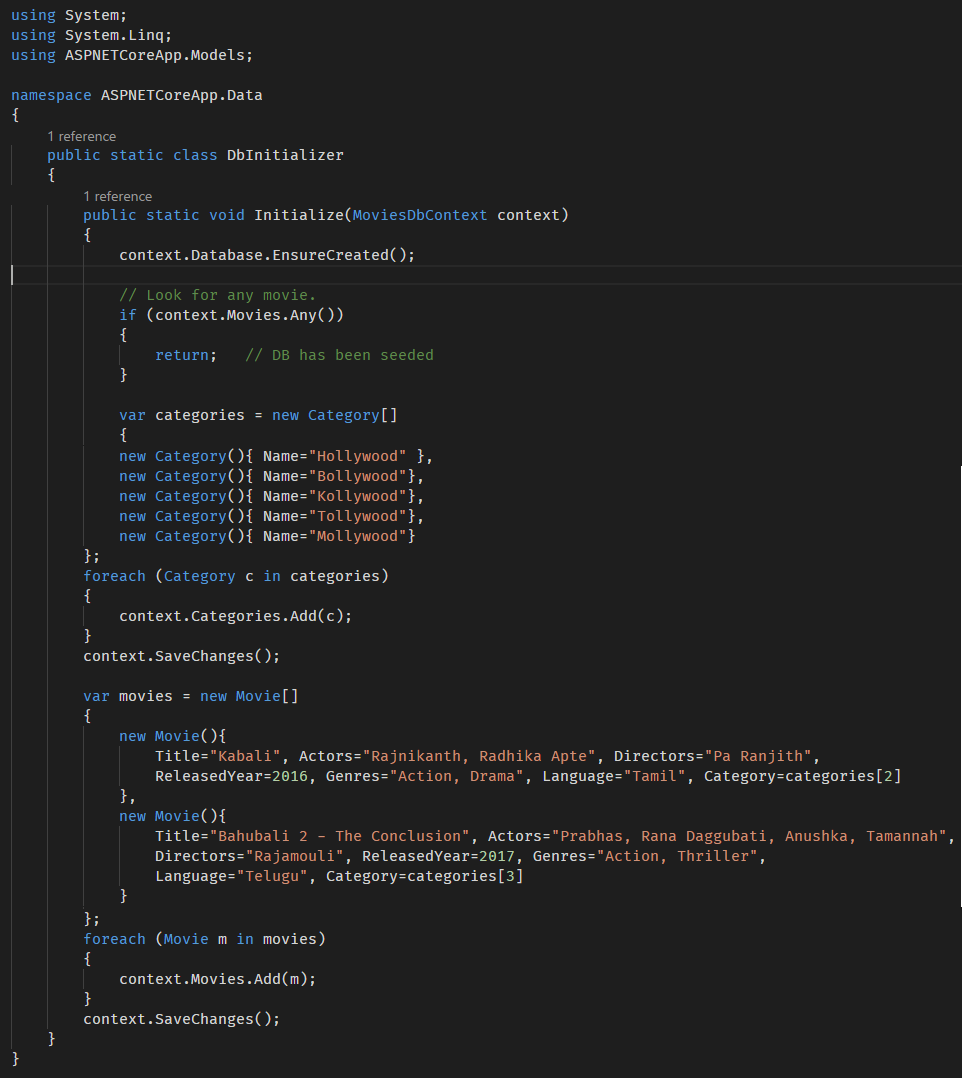
1. Run the “**dotnet ef database update**” command to generate the database.



1. Now we can create seed values for the database tables. To do so, create a new folder called “Data” in project root and create a file named “**DbInitializer.cs**” in the folder.



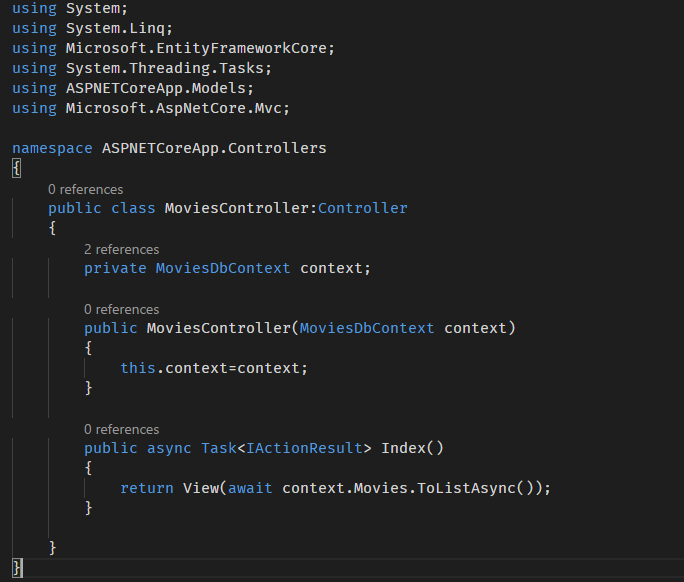
1. Add the following code ***DbInitilizer.cs*** file.



1. Now you need to call the **Initialize**() method of the **DbInitializer** class from the **Startup** class. Inject the **DbContext** class the **ConfigureServices**() method and call the **DbInitializer.Initialize()** method .



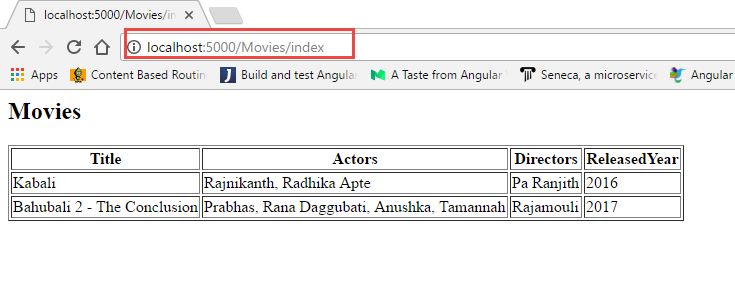
1. Run the project to insert seed values to the tables. Use the “**dotnet run**” command to run the project.
2. Create a new Controller called “**MoviesController**” inside the Controllers folder. Add the following code inside the **MoviesController**.



1. Inject the **MoviesDbContext** in the Constructor of the **MoviesController**.
2. Add the Index action and return the view to display the list of movies.
3. Add “**Movies**” folder inside the “**Views**” folder and create “**Index.cshtml**” file inside the Movies folder. Add the following code to the Index View.



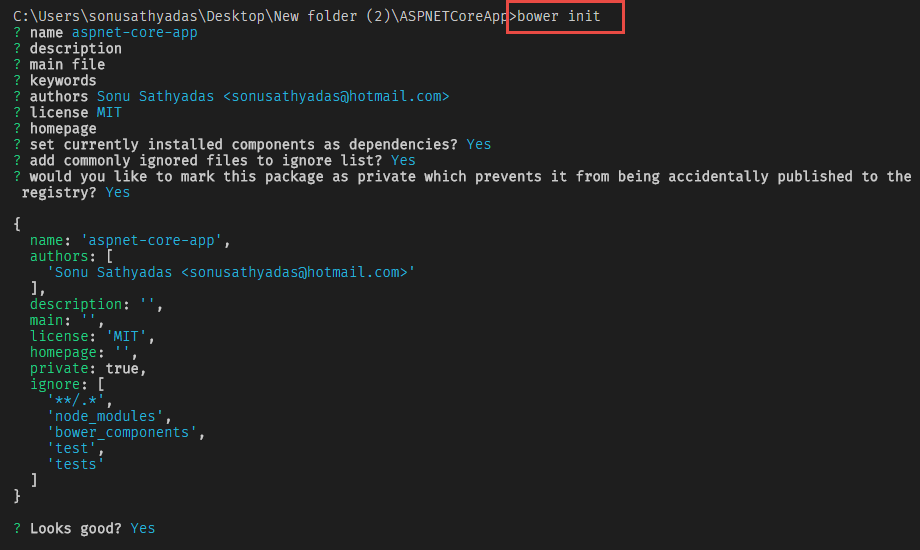
1. Run the project by executing the command “**dotnet run**”. Open the browser and navigate to the <http://localhost:5000/Movies/Index> URL.



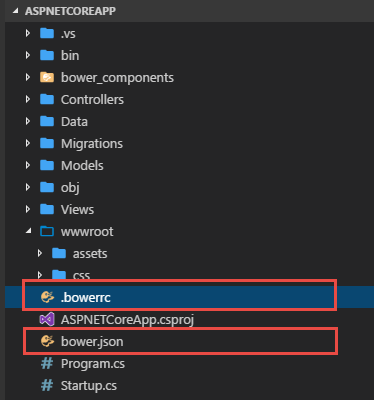
**Adding Bootstrap styling to ASP.NET Core Project**

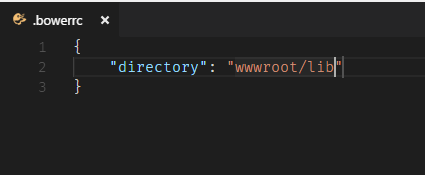
To add bootstrap to the ASP.NET core project you can use the bower tool. For that first you need to install bower tool in your system.

1. Open VS Code Terminal and run the following command, “**npm install -g bower**”.
2. Now you need to create **bower.json** file in the root folder. To create bower.json file, run the following command in the terminal, “**bower init**”.



1. You also need to create a bower settings configuration file **.bowerrc** in the root folder. By default, packages installed using bower will be stored in the **bower-components** folder. But for this project we need to store css and js files inside the **wwwroot** folder. You can change the installation directory in the .**bowerrc** file.

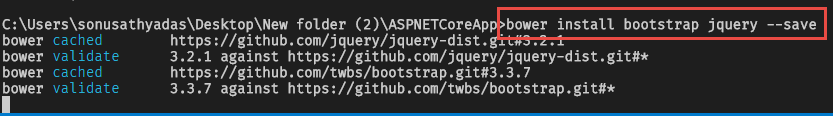




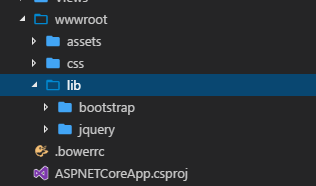
1. Now you need to run the bower commands to install the dependencies, To install bootstrap, you need to install bootstrap and jquery in the project. Run the following bower commands to install the files.

**Bower install bootstrap –save**

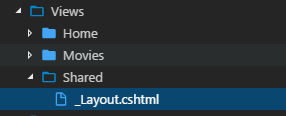
**Bower install jquery –save**



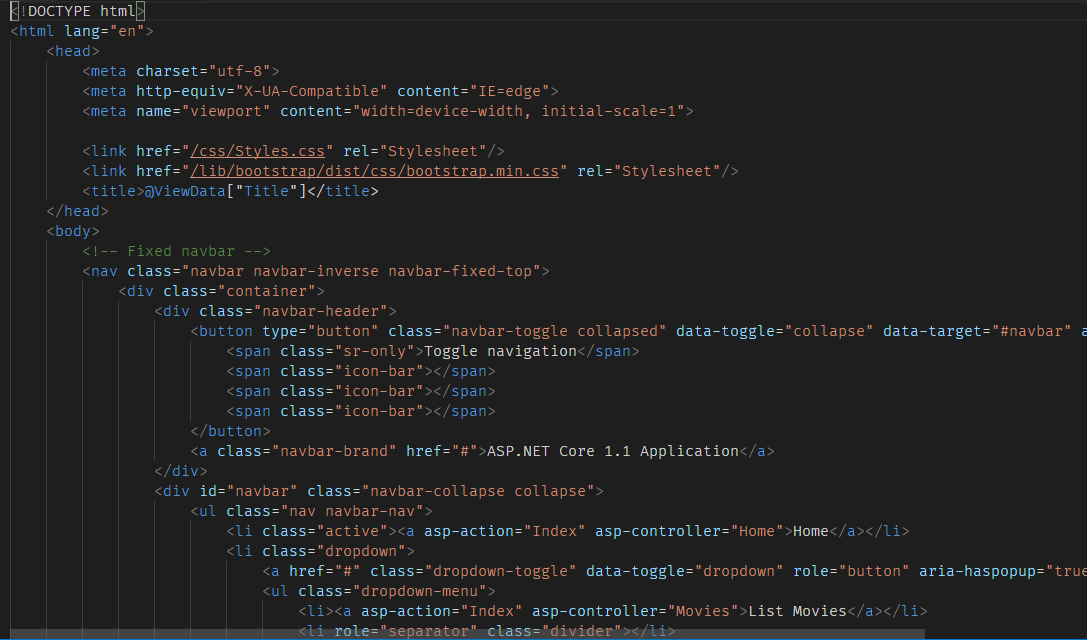
1. It adds the dependencies to the **bower.json** file. The packages will be installed inside the **wwwroot/lib** folder.



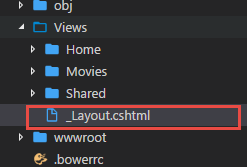
1. Now, you need to create the Layout view for the application. For that add a new file names **“\_Layout.cshtml**” in the **Views/Shared** folder. Add the bootstrap and jQuery library path’s in the layout file.

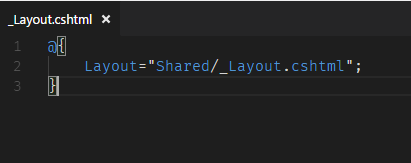


1. Go to bootstrap website and copy the bootstrap layout page code and paste in the **\_Layout.cshtml** file.

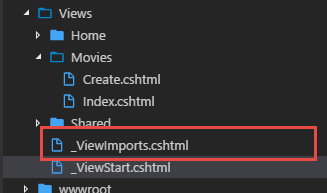


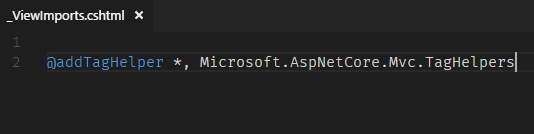
1. Now Add **\_ViewStart.cshtml** file in the **Views** folder. And add the following code within the file.



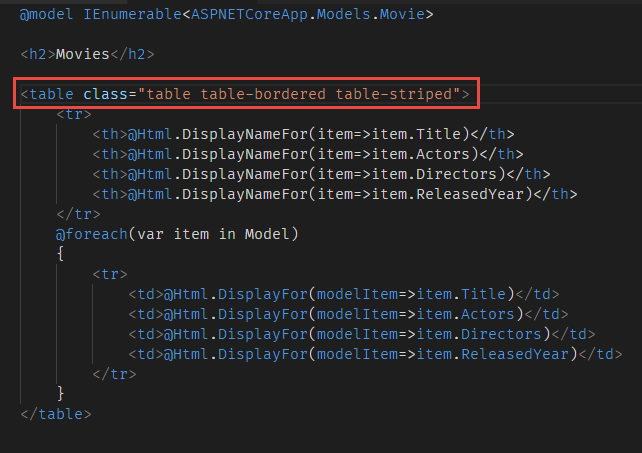


1. Also, add **\_ViewImports.cshtml** file in the **Views** folder and add the following line of code to import TagHelpers in all the view files.

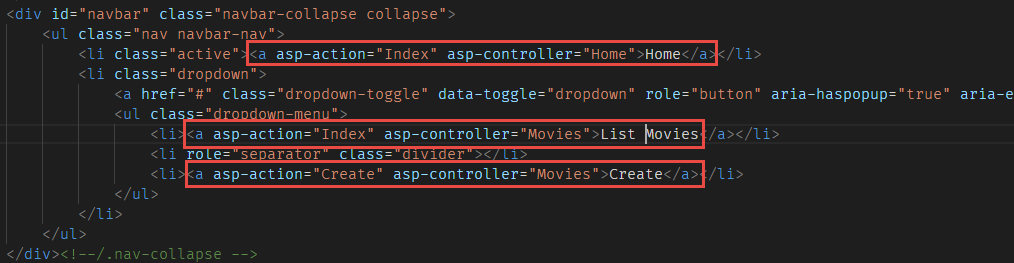




1. Add bootstrap classes to the table in the Movies>Index view.



1. Update your **\_Layout.cshtml** file to add hyperlinks to the **Home>Index** action and **Movies>Index** action. You can use anchor tag helpers to generate hyperlinks from the controller and action name.



1. Run the project and navigate to the URL <http://localhost:5000/Movies/Index>

